JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and interfaces
9. String processing
10. Exceptions and Assertions
11. Nested classes
12. Enums
13. Wrapper classes for primitive types
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java Classes
19. Object Oriented Design
20. **Functional Programing in Java**

Module contents

1. Functional Programming in Java
   - Functional Interfaces
   - Lambda Expressions
   - Predicates
   - Functions
   - Operators
   - Consumers
   - Suppliers
   - Use in Traversing Objects
   - Use in Collections
   - Use in Comparing Objects
   - Use in Optionals
   - Use in Streams

Module contents

1. Functional Programming in Java
   - Functional Interfaces
   - Lambda Expressions
   - Predicates
   - Functions
   - Operators
   - Consumers
   - Suppliers
   - Use in Traversing Objects
   - Use in Collections
   - Use in Comparing Objects
   - Use in Optionals
   - Use in Streams

Object Traversing with Iterator

The traversing elements of Collection is a frequently action in programming.

Entity class, it objects will be elements of collection

```
public class Car {
    private String make;
    private String model;
    public Car(String ma, String mo) {
        make = ma;
        model = mo;
    }
    @Override
    public String toString() {
        return make + " " + model;
    }
}
```

Object Traversing with Iterator

The Java API provides classical approach for traversing objects-elements of Collections – Iterator<E> with it methods boolean hasNext() and E next(). Java 8 add to Iterator<E> method default void forEachRemaining(Consumer<? super E> action); for traversing elements of Collections.

```
public static void main(String[] args) {
    List<Car> cars = Arrays.asList(
            new Car("Nissan", "Sentra"),
            new Car("Chevrolet", "Vega"),
            new Car("Hyundai ", "Elantra")
    );
    /* Classical Traversing */
    Iterator<Car> it = cars.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
```

Consumer

```
cars.iterator().forEachRemaining(System.out::println);
```

Object Traversing with Iterator

The forEachRemaining method of the Iterator<E> interface actually implements the while loop on the collection:

```
default void forEachRemaining(Consumer<? super E> action) {
    Objects.requireNonNull(action);      //check action for null
    while (hasNext())
        action.accept(next());
}
```

Consumer functional method invoke

Traversing Arrays of Primitive Types

An Iterator<E> can be used to traverse a Collection<E> or any object that implements the Iterable<E> interface. Is it possible to traverse a Java array in the same manner?

The **PrimitiveIterator** interface can be used to traverse a Java array of **certain** primitive types. It is generic for two types, T and T_CONS. Type T must be Integer, Long, or Double, and type T_CONS must be the corresponding specialization of Consumer.

```
public interface PrimitiveIterator<T, T_CONS> extends Iterator<T> {
    void forEachRemaining(T_CONS action);
}
```

Traversing Arrays of Primitive Types

Suppose a program needs a class that can traverse a Java array of ints. This can be accomplished by implementing the **PrimitiveIterator** interface where the first type is Integer and the second type is IntConsumer.

Wrapping array by PrimitiveIterator

```
public class IntIteratorGen implements PrimitiveIterator<Integer,
                                                         IntConsumer> {
    private int[] array;
    private int cursor;
    public IntIteratorGen(int... a) {
        cursor = 0;
        array = Arrays.copyOf(a, a.length);
    }
    @Override
    public void forEachRemaining(IntConsumer c) { //Array traversing
        while (hasNext()) {
            c.accept(array[cursor]);
            cursor++;          }      }   …
```

Traversing Arrays of Primitive Types

```
…
    @Override
    public boolean hasNext() {
        return cursor < array.length;
    }

    @Override
    public Integer next() {
        int i = 0;
        if (hasNext()) {
            i = array[cursor];
            cursor++;
        }
        return i;
    }
}
```

```
public class TestPrimitiveIteratorGen {

    public static void main(String[] args) {
        IntIteratorGen it =
                new IntIteratorGen(1, 2, 3, 4, 5);
        it.forEachRemaining((IntConsumer)x ->
                System.out.print(x + “,”));
    }
}
```

Output:
1,2,3,4,5,

Using Specializations of PrimitiveIterator

Non-generic specializations for **Integer**, **Long**, and **Double** are available as nested interfaces of **PrimitiveIterator**.

```
public static interface PrimitiveIterator.OfInt
                              extends PrimitiveIterator<Integer, IntConsumer> {
    default void forEachRemaining(IntConsumer action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(nextInt());
    }
    @Override
    default Integer next() {
        if (Tripwire.ENABLED)
            Tripwire.trip(getClass(), "{0} callingrimitiveIterator.OfInt.nextInt()");
            return nextInt();
    }
    int nextInt();    ...   }
```

PrimitiveIterator.OfLong and PrimitiveIterator.OfDouble are similar

Using Specializations of PrimitiveIterator

```
public class IntIterator implements PrimitiveIterator.OfInt {
    private int[] array;
    private int cursor;
    public IntIterator(int... a) {
        cursor = 0;
        array = Arrays.copyOf(a, a.length);
    }
    @Override
    public boolean hasNext() {
        return cursor < array.length;
    }
...
```

```
…
@Override
    public int nextInt() {
        int i = 0;
        if(hasNext()){
            i = array[cursor];
            cursor++;
        }
        return i;
    }
}
```

LongIterator and DoubleIterator are similar

Using Specializations of PrimitiveIterator

```
public class TestPrimitiveIteratorSpecialization {
    public static void main(String[] args) {
        IntIterator iit = new IntIterator(1, 2, 3, 4, 5);
        iit.forEachRemaining((IntConsumer) x -> System.out.print(x + " "));
        System.out.println();
        LongIterator lit = new LongIterator(6, 7, 8, 9, 10);
        lit.forEachRemaining((LongConsumer) x -> System.out.print(x + " "));
        System.out.println();
        DoubleIterator dit = new DoubleIterator(20.1, 21.2, 22.3, 23.4, 24.5);
        dit.forEachRemaining((DoubleConsumer) x ->
                                                System.out.print(x + " "));
        System.out.println();
    }
}
```

Output:

```
1 2 3 4 5
6 7 8 9 10
20.1 21.2 22.3 23.4 24.5
```

Traversing Objects Using Spliterators

Spliterator interface defines a default **forEachRemaining** method which accepts a consumer. Spliterator is useful for partitioning a collection into components through the use of its **trySplit** method.

```
public interface Spliterator<T> {
    default void forEachRemaining(Consumer<? super T> action) {
        do { } while (tryAdvance(action));
    };
    boolean tryAdvance(Consumer<? super T> action);
    Spliterator<T> trySplit();
...
}
```

Traversing Objects Using Spliterators

```
public class SpliIteratorTest {
    public static void main(String[] args) {
        List<Car> cars = Arrays.asList(
                new Car("Nissan", "Sentra"),
                new Car("Chevrolet", "Vega"),
                new Car("Hyundai", "Elantra"),
                new Car("Buick", "Regal")
        );
        Spliterator<Car> spliterator = cars.spliterator();
        spliterator.forEachRemaining(x ->
                                    System.out.println("In splititerator: " + x));
        spliterator = cars.spliterator();       //it need to get the iterator again
        Spliterator<Car> firstHalf = spliterator.trySplit();
        firstHalf.forEachRemaining(x -> System.out.println("In 1st half: "
                                                        + x));
        spliterator.forEachRemaining(x -> System.out.println("In 2nd half: "
                                                        + x));      }   }
```

Output:

```
In splititerator: Nissan Sentra
In splititerator: Chevrolet Vega
In splititerator: Hyundai Elantra
In splititerator: Buick Regal
In 1st half: Nissan Sentra
In 1st half: Chevrolet Vega
In 2nd half: Hyundai Elantra
In 2nd half: Buick Regal
```

Traversing Iterable Objects

In Java 8, the default **forEach** method, which accepts a consumer, was added to the Iterable interface.

```
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action) {...};
    default Spliterator<T> spliterator() {...}
}
```

```
public static void main(String[] args) {
        List<Car> cars = Arrays.asList(
                new Car("Nissan", "Sentra"),
                new Car("Chevrolet", "Vega"),
                new Car("Hyundai", "Elantra")
        );
        /*List is iterable object*/
        cars.forEach(x -> System.out.println(x));
    }
```

Output:
Nissan Sentra
Chevrolet Vega
Hyundai Elantra

Traversing Iterable Objects That Contain Arrays of Primitives

When an **Iterable** object consists of Java array of **ints**, **longs**, or **doubles**, the objects' method **Iterator<T> iterator()** can return a **PrimitiveIterator**.

```
public class MyInts implements Iterable<Integer> {
    private int[] array;
    public MyInts(int... a) {
        array = Arrays.copyOf(a, a.length);
    }
    @Override
    public PrimitiveIterator<Integer, IntConsumer> iterator() {
        return new IntIter();
    }
    ...
```

Traversing Iterable Objects That Contain Arrays of Primitives

```
...
private class IntIter implements PrimitiveIterator<Integer,
                                                   IntConsumer> {

    private int cursor;
    public IntIter() {
        cursor = 0;
    }
    @Override
    public void forEachRemaining(IntConsumer c) {
        while (hasNext()) {
            c.accept(array[cursor]);
            cursor++;
        }
    }
    ...
```

Traversing Iterable Objects That Contain Arrays of Primitives

```
...
    @Override
    public boolean hasNext() {
        return cursor < array.length;
    }
    @Override
    public Integer next() {
        int i = 0;
        if (hasNext()) {
            i = array[cursor];
            cursor++;
        }
        return i;
    }
}
}
```

```
public static void main(String[] args) {
        MyInts my = new MyInts(1, 2, 3, 4, 5);
        my.forEach(x ->
                                System.out.println(x));
        System.out.println();
        my.iterator().
      forEachRemaining((IntConsumer) x ->
                                System.out.println(x));
    }
```

Traversing Iterable Objects with Arrays of Primitives Using Specializations of PrimitiveIterator

The iterator method of the MyInts class could also have returned a **PrimitiveIterator.OfInts**. The inner class would then implement **PrimitiveIterator.OfInts** instead of **PrimitiveIterator<Integer, IntConsumer>**.

```
public class MyIntsP implements Iterable<Integer> {
    private int[] array;
    public MyIntsP(int... a) {
        array = Arrays.copyOf(a, a.length);
    }
    @Override
    public PrimitiveIterator.OfInt iterator() {
        return new IntIterP();
    }
    ...
```

Traversing Iterable Objects with Arrays of Primitives Using Specializations of PrimitiveIterator

```
...
private class IntIterP implements PrimitiveIterator.OfInt {
    int cursor;
    public IntIterP() {
        cursor = 0;
    }
    @Override
    public boolean hasNext() {
        return cursor < array.length;
    }
...
```

Traversing Iterable Objects with Arrays of Primitives Using Specializations of PrimitiveIterator

```
...
        @Override
        public int nextInt() {
            int i = 0;
            if (hasNext()) {
                i = array[cursor];
                cursor++;
            }
            return i;
        }
    }
}
```

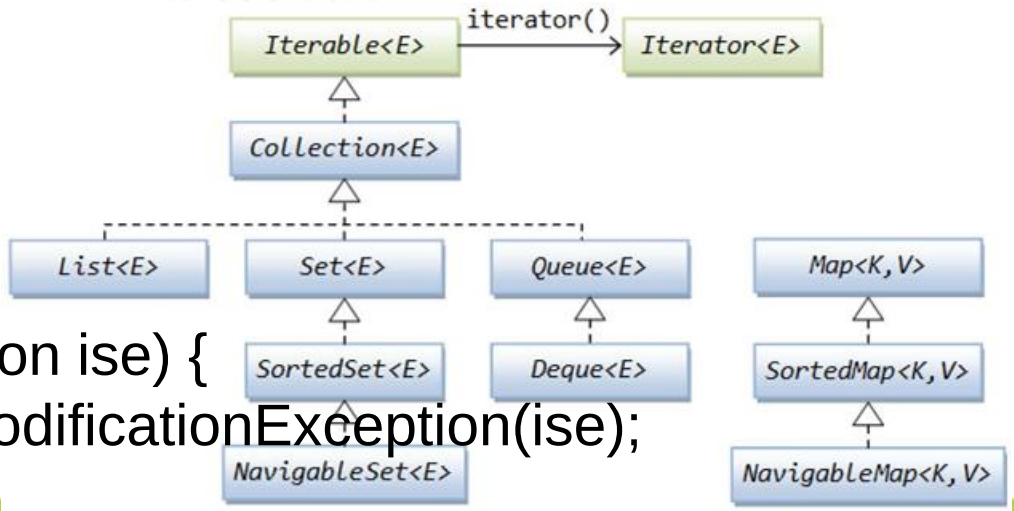
```
public static void main(String[] args) {

        MyIntsP my = new MyIntsP(1, 2, 3, 4, 5);
        my.forEach(x -> System.out.println(x));
        System.out.println();
        my.iterator().
            forEachRemaining((IntConsumer) x ->
                                    System.out.println(x));
    }
```

PrimitiveIterator.OfLong and PrimitiveIterator.OfDouble are similar

Traversing Map

Java maps are not iterable. However, a **forEach** method has been provided for traversal. Unlike Iterable objects with **`default void forEach(Consumer<? super T> action)`** method, the **forEach** method of the Map interface accepts a BiConsumer object which processes the map entry's key and value.

```
default void forEach(BiConsumer<? super K,? super V> action) {
    Objects.requireNonNull(action);
    for (Map.Entry<K, V> entry : entrySet()) {
        K k;
        V v;
        try {
            k = entry.getKey();
            v = entry.getValue();
        } catch (IllegalStateException ise) {
            throw new ConcurrentModificationException(ise);
        }
        action.accept(k, v);      }    }
```



Traversing Map

```
public static void main(String[] args) {
    Map<String, Double> employeeSalaries = new TreeMap<>();
    employeeSalaries.put("Joe Smith", 100000.0);
    employeeSalaries.put("Maggie Jones", 110000.0);
    employeeSalaries.put("Larry Rodriguez", 105000.0);
    /*forEach method accepts a BiConsumer */
    employeeSalaries.forEach((x, y) -> System.out.println(x
                                        + " makes $" + y + " annually."));
    }
```

Output:

Joe Smith makes $100000.0 annually.
Larry Rodriguez makes $105000.0 annually.
Maggie Jones makes $110000.0 annually.

Traversing Set

Since a Java Set is iterable, to the **forEachRemaining** method of its **Iterator<E>** can be passed a consumer.
The method **forEach** of **Iterable<E>** can also be used.

```
public static void main(String[] args) {
    Set<String> colors = new TreeSet<>();
    colors.add("red");
    colors.add("green");
    colors.add("blue");
    colors.iterator().forEachRemaining(x -> System.out.print(x + " "));
    System.out.println();
    colors.forEach(x -> System.out.print(x + " "));
    System.out.println();
}
```

Output:
blue green red
blue green red

Module contents

1. Functional Programming in Java
   - Functional Interfaces
   - Lambda Expressions
   - Predicates
   - Functions
   - Operators
   - Consumers
   - Suppliers
   - Use in Traversing Objects
   - Use in Collections
   - Use in Comparing Objects
   - Use in Optionals
   - Use in Streams

Removing Elements from a Collection

The **removeIf** method can be used to remove elements from a Collection<E> implementation's object whose Iterator supports the remove operation.

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

In Collection<E> interface

```
public interface Iterator<E> {
...
    default void remove() {
        throw new
         UnsupportedOperationException("remove");
    }
}
```

Removing Elements from a Collection

If the predicate supplied to the **removeIf** method is true for an element, that element is removed from the collection.

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("Super");
    list.add("Random");
    list.add("Silly");
    list.add("Strings");
    list.removeIf(x -> x.startsWith("S"));
    list.forEach(x -> System.out.println(x));      //Random
    Set<String> names = new TreeSet<>();
    names.add("Jeremy");
    names.add("Javier");
    names.add("Rose");
    names.removeIf(x -> x.charAt(0) == 'J');
    names.forEach(x -> System.out.println(x));      //Rose   }
```

Populating an Array

The **Arrays** class has several **setAll** methods that will populate each element of an array passed as the first argument using an operator or function passed as the second argument. ***The first parameter of the operator or function is the subscript corresponding to the array element.***

static <T> void **setAll**(T[] array, IntFunction<? extends T>
generator)
static void **setAll**(int[] array, IntUnaryOperator generator)
static void **setAll**(long[] array, IntToLongFunction generator)
static void **setAll**(double[] array, IntToDoubleFunction generator)

java.utils.Arrays

Populating an Array

```
public static void main(String[] args) {
    /*Sets each element of int array iarr equal to its subscript (index).*/
    IntUnaryOperator iop = x -> x;
    int[] arr = new int[4];
    Arrays.setAll(arr, iop);
    for (int i : arr) {
        System.out.println(i);          //0 1 2 3
    }
    /*Sets each element of long array larr equal to 5.*/
    IntToLongFunction gen5 = x -> 5;
    long[] larr = new long[4];
    Arrays.setAll(larr, gen5);
    for (long l : larr) {
        System.out.println(l);
    }
    …
```

Populating an Array

```
…
/*Sets each element of double array darr equal to a random number
   between 0.0 and 1.0.*/
IntToDoubleFunction i2d = x -> (new Random()).nextFloat();
double[] darr = new double[4];
Arrays.setAll(darr, i2d);
for (double d : darr) {
    System.out.println(d);
}
/*Populates an array of strings such that each element contains the
  letter "S" repeated the number of subscript times.*/
IntFunction<String> is = x -> {
    String s = "";
    for (int i = 0; i <= x; i++) {
        s += "S";
    }
    return s;
}; …
```

```
…
String[] sarr = new String[4];
Arrays.setAll(sarr, is);
for (String s : sarr) {
    System.out.println(s);
}
}
```

Output:

S
SS
SSS
SSSS

Replacing the Elements of a List or a Map

All the elements in a list can be modified using the default **replaceAll** method and a UnaryOperator that specified how to perform the modification.

```
default void replaceAll(UnaryOperator<E> operator) {
        Objects.requireNonNull(operator);
        final ListIterator<E> li = this.listIterator();
        while (li.hasNext()) {
            li.set(operator.apply(li.next()));
        }
    }
```

All the element in a map can also be modified using the default **replaceAll** method and a BiFunction.

```
default void replaceAll(BiFunction<? super K, ? super V,
                                   ? extends V> function) {
    Objects.requireNonNull(function);
…
```

Replacing the Elements of a List or a Map

```
...
    for (Map.Entry<K, V> entry : entrySet()) {
        K k;
        V v;
        try {
            k = entry.getKey();
            v = entry.getValue();
        } catch (IllegalStateException ise) {
            // this usually means the entry is no longer in the map.
            throw new ConcurrentModificationException(ise);          }
        // ise thrown from function is not a cme.
        v = function.apply(k, v);
        try {
            entry.setValue(v);
        } catch (IllegalStateException ise) {
            // this usually means the entry is no longer in the map.
            throw new ConcurrentModificationException(ise);
        }       }    }
```

Replacing the Elements of a List or a Map

```
public static void main(String[] args) {
    List<Integer> list = Arrays.asList(16, 12, 8, 4);
    UnaryOperator<Integer> div4 = x -> x / 4;
    list.replaceAll(div4);
    list.forEach(x -> System.out.print(x + " "));
    System.out.println();

    Map<String, String> map = new TreeMap<>();
    map.put("Smith", "Robert");
    map.put("Jones", "Alex");
    BiFunction<String, String, String> bi = (k,v) -> "Mr. " + v;
    map.replaceAll(bi);
    map.forEach((k,v) -> System.out.println(v + " " + k));
}
```

Output:

4 3 2 1

Mr. Alex Jones

Mr. Robert Smith

Parallel Computations on Arrays

The Arrays class has several **parallelPrefix** methods that perform parallel computations on the elements of an array or on a subrange of the elements in the array. The methods accept the **array** and a **binary operator** that specifies the computation to be performed.

static void parallelPrefix(**double[]** array, int fromIndex, int toIndex, **DoubleBinaryOperator** op);
static void parallelPrefix(**double**[] array, **DoubleBinaryOperator** op);
static void parallelPrefix(**int[]** array, int fromIndex, int toIndex, **IntBinaryOperator** op);
static void parallelPrefix(**int[]** array, **IntBinaryOperator** op);
static void parallelPrefix(**long**[] array, int fromIndex, int toIndex, **LongBinaryOperator** op);
static void parallelPrefix(**long**[] array, **LongBinaryOperator** op);
static void parallelPrefix(**T[]** array, int fromIndex, int toIndex, **BinaryOperator<T>** op);
static void parallelPrefix(**T[]** array, **BinaryOperator<T>** op);

Parallel Computations on Arrays

```
public class ArraysParallelComputations {
    public static void main(String[] args) {
        int[] arr = {2,3,4,3};
        IntBinaryOperator op = (x,y) -> x*y;
        Arrays.parallelPrefix(arr, op);
        for (int i : arr) {
            System.out.println(i);
        }
    }
}
```

Output:
2
6
24
72

The Arrays class has several **parallelSetAll** methods with the same arguments as the **setAll** methods, but executed in parallel

Map Computations

The Map interface provides the following methods which perform inline computations on an entry in a map. The Map interface's default compute method performs its computation on an entry in a map using the specified **BiFunction**. If the **BiFunction** results in null, the entry is removed from the map.

```
default V compute<K key, BiFunction<? super K, ? super V,
                                    ? extends V> remappingFunction);
default V computeIfAbsent<K key, Function<? super K,
                                    ? extends V> mappingFunction);
default V computeIfPresent<K key, BiFunction<? super K, ? super V,
? extends V> remappingFunction);
```

computeIfAbsent method accepts a **Function** object instead of a BiFunction, since no existing value needs to be processed

Map Computations

```
public static void main(String[] args) {
    Map<String, Integer> map = new TreeMap<>();
    map.put("RED", 32);
    map.put("GREEN", null);

    /* The compute method performs calculations defined by the
       BiFunction object for entries. If BiFunction returns null, the entry
       is deleted.*/
    BiFunction<String, Integer, Integer> bin = (k, v) -> v == null ?
                                                        null : v / 4;
    System.out.println(map.compute("RED", bin));            //8
    System.out.println(map.compute("GREEN", bin));          //null – entry
                                                             //deleted
    System.out.println(map.compute("YELLOW", bin));         //null –
                                                    //nothing is done
    map.forEach((k, v) -> System.err.println(k + " " + v));    //RED 8
    ...
```

Map Computations

```
....
   /* The computeIfPresent method performs calculations for entries
      defined by the BiFunction object, if the entry exists and its value
      is not null (nothing is done for the value of null). If BiFunction
      returns null, the entry is deleted..
      Recovering a deleted entry */
   map.put("GREEN", null);
   /*BiFunction never returns null*/
   BiFunction<String, Integer, Integer> bi = (k, v) -> v / 4;
   System.out.println(map.computeIfPresent("RED", bin));          //2
   System.out.println(map.computeIfPresent("GREEN", bin)); //null –
                                                          //no deleted
   System.out.println(map.computeIfPresent("YELLOW", bin)); //null –
                                                          //nothing is done
   map.forEach((k, v) -> System.err.println(k + " " + v)); //GREEN null,
                                                            //RED 2
...
```

Map Computations

```
...
    /*The computeIfAbsent method performs calculations for entries
      defined by the Function object if the entry is missing or if the
      entry value is null. If the function returns null, no entry is added.
      It accepts a Function object instead of a BiFunction, since no
      existing value needs to be processed.*/
      Function<String, Integer> fi = k -> k.length();
      Function<String, Integer> finull = k -> null;
      System.out.println(map.computeIfAbsent("RED", fi));    //2
                                                    //nothing is done
      System.out.println(map.computeIfAbsent("GREEN", fi));      //5
                                                //(number of characters in the key)
      System.out.println(map.computeIfAbsent("YELLOW", fi));     //6
                                                //(number of characters in the key)
      System.out.println(map.computeIfAbsent("BLACK", finull)); //null
                                                    //(does not add an entry)
      map.forEach((k, v) -> System.err.print(k + " " + v + ", "));
                                                //GREEN 5, RED 2, YELLOW 6,
    }
```

Map Merging

The Map interface's default **merge** method is mainly used to modify an existing value by merging portions of a new value with it according to a mapping function. If the entry does not exist, a new entry is created with the specified key and value. If the mapping function results in a null value, the entry is removed from the map. The new value is not allowed to be null.

```
default V merge(K key, V value, BiFunction<? super V, ? super V,
                                          ? extends V> remappingFunction) {
    Objects.requireNonNull(remappingFunction);
    Objects.requireNonNull(value);
    V oldValue = get(key);
    V newValue = (oldValue == null) ? value :
             remappingFunction.apply(oldValue, value);
    if (newValue == null) {
        remove(key);
    } else {
        put(key, newValue);        }        return newValue;    }
```

Map Merging

```
public class MyClass {
    int i1;
    int i2;
    String s;
    public MyClass(int x, int y, String z) {
        i1 = x;
        i2 = y;
        s = z;
    }
    @Override
    public String toString() {
        return i1 + " " + i2 + " " + s;
    }
}
```

Map Merging

```
public static void main(String[] args) {
    Map<String, MyClass> m = new TreeMap<>();
    m.put("k1", new MyClass(1, 2, "Dog"));
    BiFunction<MyClass, MyClass, MyClass> changeI2 = (ov, nv)
                                        -> new MyClass(ov.i1, nv.i2, ov.s);
    BiFunction<MyClass, MyClass, MyClass> changeS = (ov, nv)
                                        -> new MyClass(ov.i1, ov.i2, nv.s);
    System.out.println(m.merge("k1", new MyClass(0, 5, null),
                                                changeI2));   //1 5 Dog
    System.out.println(m.merge("k1", new MyClass(0, 0, "Cat"),
                                                changeS));   //1 5 Cat
    /*No key “k2” in the map - entry added*/
    System.out.println(m.merge("k2", new MyClass(6, 7, "Bird"),
                                                changeS));   //6 7 Bird
    m.forEach((k, v) -> System.err.print(k + " " + v + ", "));
                                              //k1 1 5 Cat, k2 6 7 Bird,
    }
```

Functional Interfaces and Sets

Since the **Set** interface supports the remove operation, the **removeIf** method can be used to remove elements from a set. If the supplied predicate is true for an element, that element is removed from the set.

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

In Collection<E> interface

Functional Interfaces and Sets

```
public static void main(String[] args) {
    Set<String> names = new TreeSet<>();
    names.add("Jeremy");
    names.add("Javier");
    names.add("Rose");

    names.removeIf(x -> x.charAt(0) == 'J');
    names.forEach(System.out::println);    //Rose
}
```

Module contents

1. Functional Programming in Java
   - Functional Interfaces
   - Lambda Expressions
   - Predicates
   - Functions
   - Operators
   - Consumers
   - Suppliers
   - Use in Traversing Objects
   - Use in Collections
   - Use in Comparing Objects
   - Use in Optionals
   - Use in Streams

Comparator Interface

Comparator is a functional interface that is used to compare two objects. A Comparator is specified with type parameter T. Its functional method, called **compare**, takes two arguments of type T and returns an integer.

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
    ...
}
```

The result of the compare method is as follows:

- a positive integer: if o1 > o2
- a negative integer: if o1 < o2
- 0: if o1 = o2

Comparator Interface

```
public class Comparing {
    public static String removeVowels(String s) {
        return s.replaceAll("[aeiou]", "");
    }
    public static void main(String[] args) {
        Comparator<String> byConsonants = (x, y) ->
                                removeVowels(x).compareTo(removeVowels(y));
        System.out.println(byConsonants.compare("Larry", "Libby")); //16
        Comparator<Integer> byIntCompareTo = (x, y) ->
                                                    x.compareTo(y);
        System.out.println(byIntCompareTo.compare(1000, 1002));    //-1
                                    //because Integer compareTo realization
        Comparator<Integer> byIntDifference = (x, y) -> x - y;
        System.out.println(byIntDifference.compare(1000, 1002));     //-2
    }
}
```

Define Comparators by Function

Comparator Methods

The following Comparator methods are useful:

```
static<T extends Comparable<? super T>> Comparator<T>
                                                    naturalOrder();
static<T extends Comparable<? super T>> Comparator<T>
                                                    reverseOrder();
/*To prevent a NullPointerException while passing a null to the
compare method use the nullsFirst method which creates a new
comparator that treats nulls as being less than non-null objects*/
static <T> Comparator<T> nullsFirst(Comparator<? super T>
                                                        comparator);
static <T> Comparator<T> nullsLast(Comparator<? super T>
                                                        comparator);
default Comparator<T> reversed();
```

Comparator Methods

```
public static void main(String[] args) {
    Comparator<String> natural = Comparator.naturalOrder();
    System.out.println(natural.compare("Larry", "Libby"));      //-8
    Comparator<String> reversed = Comparator.reverseOrder();
    System.out.println(reversed.compare("Larry", "Libby"));     //8
    /*To prevent a NullPointerException while passing a null to the
      compare method use the nullsFirst method which creates a new
      comparator that treats nulls as being less than non-null objects*/
    Comparator<String> byConsonants = (x, y)
            -> removeVowels(x).compareTo(removeVowels(y));
    System.out.println(Comparator.nullsFirst(byConsonants)
                                          .compare("Larry", null));     //1
    System.out.println(Comparator.nullsLast(byConsonants)
                                          .compare("Larry", null));     //-1
    /*A new comparator compares consonants in reverse order*/
    System.out.println(byConsonants.reversed().compare("Larry",
                                          "Libby"));        //-16
} }
```

Comparator Methods

```
/*Accepts a function that extracts a  Comparable sort key from a type T, and
returns a Comparator<T> that compares by that sort key*/
public static <T, U extends Comparable<? super U>> Comparator<T>
comparing(Function<? super T, ? extends U> keyExtractor) {
    Objects.requireNonNull(keyExtractor);
        return (Comparator<T> & Serializable)
            (c1, c2) ->
                keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
    }
/*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*/
public static <T, U> Comparator<T> comparing(
            Function<? super T, ? extends U> keyExtractor,
            Comparator<? super U> keyComparator) {
        Objects.requireNonNull(keyExtractor);
        Objects.requireNonNull(keyComparator);
        return (Comparator<T> & Serializable)
            (c1, c2) -> keyComparator.compare(keyExtractor.apply(c1),
                                              keyExtractor.apply(c2));      }
```

Comparator Methods

```
public class Student {

    String name;
    Integer id;
    Double gpa;

    public Student(String n, int i, double g) {
        name = n;
        id = i;
        gpa = g;
    }

    @Override
    public String toString() {
        return name + " " + id + " " + gpa;
    }
}
```

Comparator Methods

keyExtractor

```
public static void main(String[] args) {
    Student s1 = new Student("Larry", 1000, 3.82);
    Student s2 = new Student("Libby", 1001, 3.76);
    /*For Students comparing by GPA*/
    Function<Student, Double> gpaKey = x -> x.gpa;
    Comparator<Student> byGpa=Comparator.comparing(gpaKey);
    System.out.println(byGpa.compare(s1, s2));                    //1
    /*For Students comparing by id*/
    Comparator<Student> byId = Comparator.comparing(x -> x.id);
    System.out.println(byId.compare(s1, s2));                     //-1
    /*For Students comparing by name*/
    Comparator<Student> byName = Comparator.comparing(x ->
                                                        x.name);
    System.out.println(byName.compare(s1, s2));                   //-8
    …
```

Comparator Methods

keyExtractor, keyComparator

```
…
/*For Students comparing by name using consonants only*/
Comparator<Student> byNameConsonants =
        Comparator.comparing(x -> x.name, (x, y) ->
                        removeVowels(x).compareTo(removeVowels(y)));
System.out.println(byNameConsonants.compare(s1, s2));      //16
/*For Students comparing by GPA using it ceiling*/
Comparator<Student> byGpaCeil = Comparator.comparing(x ->
                        x.gpa, (x, y) -> (int) (Math.ceil(x) - Math.ceil(y)));
System.out.println(byGpaCeil.compare(s1, s2));                    //0
//(because both 3.82 and 3.76 have a ceiling of 4.0)
…
```

```
public class ListWrapper {
    List<Integer> list;
    public ListWrapper(Integer... i) {
        list = Arrays.asList(i);
    }
}
```

Comparator Methods

```
    …
    ListWrapper list1 = new ListWrapper(2, 4, 6);
    ListWrapper list2 = new ListWrapper(1, 3, 5);
    /*ERROR: list not comparable*/
//    Comparator<ListWrapper> byList = Comparator.comparing(x ->
                                                          x.list);
    Comparator<List<Integer>> byElement0 = (x, y) ->
                                              x.get(0).compareTo(y.get(0));
    Comparator<ListWrapper> byList = Comparator.comparing(x ->
                                                x.list, byElement0);
    System.out.println(byList.compare(list1,list2));                  //1
}
```

Comparator's Methods Specialization

The Java API provides specializations to the Comparator's **comparing** method that extract **Doubles**, **Integers**, and **Long** keys from objects before comparing them based on natural ordering.

```
static <T> Comparator<T>
comparingDouble(ToDoubleFunction<? super T>keyExtractor);
static <T> Comparator<T>
comparingInt(ToIntFunction<? super T>keyExtractor);
static <T> Comparator<T>
comparingLong(ToLongFunction<? super T>keyExtractor);
```

```
public class LongWrapper {
    Long l;
    public LongWrapper(long a) {
        l = a;
    }
}
```

Comparator's Methods Specialization

```
public static void main(String[] args) {
    Student s1 = new Student("Larry", 1000, 3.82);
    Student s2 = new Student("Libby", 1001, 3.76);
    ToDoubleFunction<Student> gpaKey2 = x -> x.gpa;    //keyExtractor
    System.out.println(Comparator.comparingDouble(gpaKey2)
                                 .compare(s1, s2));              //1
    / * Comparison of integers based on the natural order * /
//       System.out.println(Comparator.comparingInt(x ->
x.id).compare(s1, s2)); // ERROR
    System.out.println(Comparator.comparingInt((Student x) -> x.id)
                                 .compare(s1, s2));              //-1
    LongWrapper l1 = new LongWrapper(4L);
    LongWrapper l2 = new LongWrapper(4L);
    ToLongFunction<LongWrapper> lKey = x -> x.l;
    System.out.println(Comparator.comparingLong(lKey)
                                 .compare(l1, l2));              //0
}
```

Comparators Chains

The default **thenComparing** methods return a comparator that is used for further comparison if the calling comparator determines that the objects being compared are equal.

```
default Comparator<T> thenComparing(Comparator<? super T> other);
default <U extends Comparable<? super U>> Comparator<T>
    thenComparing(Function<? super T, ? extends U> keyExtractor);
default <U> Comparator<T> thenComparing(Function<? super T,
 ? extends U> keyExtractor, Comparator<? super U> keyComparator);
```

Comparators Chains

```
public static void main(String[] args) {
    Student s1 = new Student("Joseph", 1000, 3.82);
    Student s2 = new Student("Joseph", 1002, 3.82);
    Comparator<Student> byName = Comparator.comparing(x ->
                                                        x.name);
    Comparator<Student> byId = Comparator.comparing(x -> x.id);
    Comparator<Student> byGpa = Comparator.comparing(x -> x.gpa);
    System.out.println(byName.compare(s1, s2));                  //0
    System.out.println(byName.thenComparing(byGpa)
                              .compare(s1, s2));            //byName->byGpa  0
    System.out.println(byName.thenComparing(byId)
      .thenComparing(byGpa)
      .compare(s1, s2));  //byName->byId->byGpa   -1 while id compare
     System.out.println(byName.thenComparing(byGpa)
      .thenComparing(byId)
      .compare(s1, s2));  //byName->byGpa->byId   -1 while id compare
```

,,,

Comparators Chains

```
,,,
Comparator<Student> byNameConsonants =
        Comparator.comparing(x -> x.name,
        (x, y) -> removeVowels(x).compareTo(removeVowels(y)));
    Comparator<Integer> byDifference = (x, y) -> x - y;
    Comparator<Double> byCeil = (x, y)
                                    -> (int) (Math.ceil(x) - Math.ceil(y));

    Student s3 = new Student("Jean", 1003, 3.86);
    Student s4 = new Student("Jen", 1005, 3.69);
    System.out.println(byNameConsonants
        .thenComparing(x -> x.gpa, byCeil)
        .thenComparing(x -> x.id, byDifference)
        .compare(s3, s4));          //byName->byGpa->byId     -2

}
```

Specializing Comparators Chains

The Java API provides specializations to the Comparator interface's **thenComparing** signature that accepts a specialized function object. These specializations extract **Doubles**, **Integers**, and **Long** keys from objects before comparing them based on natural ordering.

```
default Comparator<T> thenComparingInt(ToIntFunction<? super T>
                                       keyExtractor);
default Comparator<T> thenComparingLong(ToLongFunction<?
                                        super T> keyExtractor);
default Comparator<T> thenComparingDouble(ToDoubleFunction<?
                                          super T> keyExtractor);
```

Specializing Comparators Chains

```
public static void main(String[] args) {
    Student s5 = new Student("Kaitlyn", 1006, 3.69);
    Student s6 = new Student("Jane", 1007, 3.69);
    Comparator<Student> byName = Comparator.comparing(x ->
                                                        x.name);
    Comparator<Student> byGpa = Comparator.comparing(x -> x.gpa);
    Comparator<Student> byId = Comparator.comparing(x -> x.id);
    ToIntFunction<Student> byIdfunc = x -> x.id;
    System.out.println(byGpa.thenComparingInt(byIdfunc)
                            .compare(s5, s6));                  //-1
    Student s7 = new Student("Robert", 1008, 3.86);
    Student s8 = new Student("Robert", 1009, 3.69);
//    ToDoubleFunction<Student> byGpafunc = x -> x.gpa;
    System.out.println(byName.thenComparingDouble(x -> x.gpa)
                            .compare(s7, s8));                   //1
}
```

Using Comparators to Sort Lists

Comparators will most frequently be used to sort lists and streams. The List<X> interface has a method named **sort** which accepts a Comparator<X> argument that is used for the comparisons during the sort.

```
public static void main(String[] args) {
    List<Student> students = Arrays.asList(
            new Student("Joseph", 1623, 3.54),
            new Student("Annie", 1923, 2.94),
            new Student("Sharmila", 1874, 1.86),
            new Student("Harvey", 1348, 1.78),
            new Student("Philipp", 1004, 3.90),
            new Student("Annie", 1245, 2.87)
    );
    Comparator<Student> byGpaCeil = Comparator.comparing(x ->
                            x.gpa, (x, y) -> (int) (Math.ceil(x) - Math.ceil(y)));
    students.sort(byGpaCeil);
    students.forEach(x -> System.out.println(x));          //byGpaCeil …
```

Using Comparators to Sort Lists

```
…
 students.sort(byGpaCeil.thenComparing(x -> x.name));
 students.forEach(x -> System.out.println(x));  //byGpaCeil->byName

 students.sort(byGpaCeil
        .thenComparing(x -> x.id)
        .thenComparing(x -> x.name));
 students.forEach(x -> System.out.println(x));   //byGpaCeil->byId
                                                 //->byName
}
```

Output:

```
Sharmila 1874 1.86
Harvey 1348 1.78
Annie 1923 2.94
Annie 1245 2.87
Joseph 1623 3.54
Philipp 1004 3.9
```

```
Harvey 1348 1.78
Sharmila 1874 1.86
Annie 1923 2.94
Annie 1245 2.87
Joseph 1623 3.54
Philipp 1004 3.9
```

```
Harvey 1348 1.78
Sharmila 1874 1.86
Annie 1245 2.87
Annie 1923 2.94
Philipp 1004 3.9
Joseph 1623 3.54
```

Using Comparators to Sort Java Arrays

Comparators can operate on Java arrays in a fashion similar to lists through the use of the Arrays class.

```
public static void main(String[] args) {
    Student[] students = {
        new Student("Joseph", 1623, 3.54),
        new Student("Annie", 1923, 2.94),
        new Student("Sharmila", 1874, 1.86),
        new Student("Harvey", 1348, 1.78),
        new Student("Philipp", 1004, 3.90),
        new Student("Annie", 1245, 2.87)
    };
    Student[] studentsCopy = Arrays.copyOf(students, students.length);
    Comparator<Student> byGpaCeil = Comparator.comparing(x ->
                                x.gpa, (x, y) -> (int) (Math.ceil(x) - Math.ceil(y)));
    Arrays.sort(students, byGpaCeil.thenComparing(x -> x.id)
                .thenComparing(x -> x.name));
    ...
```

Using Comparators to Sort Java Arrays

```
...
for (Student student : students) {
    System.out.println(student);          //byGpaCeil->byId->byName
}
Arrays.sort(studentsCopy, 2, 5, Comparator.comparing(x ->x.name));
final int NUM_STUDENTS = 1000;
NumberFormat fmt = NumberFormat.getNumberInstance();
fmt.setMaximumIntegerDigits(3);
Student[] studentBody = new Student[NUM_STUDENTS];
for (int i = 0; i < NUM_STUDENTS; i++) {
    studentBody[i] = new Student("S" + fmt.format(i), i, 0.0);
}
int index = Arrays.binarySearch(studentBody, new Student("S647",
                                                          0, 0.0),
Comparator.comparing(x -> x.name));
System.out.println("index = " + index + " "
        + studentBody[index]);              //index = 647 S647 647 0.0
}
```

Using Comparators to Organize Maps

Comparators can also be used to compare Map.

```
public static void main(String[] args) {
    Comparator<String> byConsonants = (x, y) -> removeVowels(x)
                                    .compareTo(removeVowels(y));
    TreeMap<String, String> pets = new TreeMap<>(byConsonants);
    pets.put("gerbil", "small cute rodents");
    pets.put("guinea pig", "rodents, not pigs");
    pets.put("cat", "have nine lives");
    pets.put("chicken", "more populous than people");
    pets.forEach((x, y) -> System.out.println(x + ", " + y));
    Comparator<Map.Entry<String, String>> cmap =
                                                Map.Entry.comparingByKey();
    Map.Entry<String, String> cat = pets.ceilingEntry("cat");
    Map.Entry<String, String> chicken = pets.ceilingEntry("chicken");
    System.out.println(cmap.compare(cat, chicken));          //-7
    ....
```

Using Comparators to Organize Maps

```
    ....
    Comparator<Map.Entry<String, String>> cmapCons =
                                Map.Entry.comparingByKey(byConsonants);
    System.out.println(cmapCons.compare(cat, chicken));        //12
    Comparator<Map.Entry<String, String>> cval
            = Map.Entry.comparingByValue();
    System.out.println(cval.compare(cat, chicken));            //-5
}
```

Using Comparators in BinaryOperator Methods

The BinaryOperator<T> **maxBy** and **minBy** methods compare two objects of type X based on a Comparator<T>

```
public static void main(String[] args) {
    Comparator<Integer> abscompare = Comparator.comparing(x ->
                                                    Math.abs(x));
    BinaryOperator<Integer> bigint = BinaryOperator
                                                .maxBy(abscompare);
    BinaryOperator<Integer> smallint = BinaryOperator
                                                .minBy(abscompare);
    System.out.println(bigint.apply(2, -5));          //-5
    System.out.println(smallint.apply(2, -5));         //2
}
```

Module contents

1. Functional Programming in Java
   - Functional Interfaces
   - Lambda Expressions
   - Predicates
   - Functions
   - Operators
   - Consumers
   - Suppliers
   - Use in Traversing Objects
   - Use in Collections
   - Use in Comparing Objects
   - Use in Optionals
   - Use in Streams

Creating an Optional

Optional is a container object which may or may not contain a non-null value. It is useful to wrap an object inside an Optional to avoid checking for nullness. Optionals can also be used inside method chains to simplify the logic of a program.
The Optional class wraps an object of type parameter T.

```
public final class Optional <T> {
    private static final Optional<?> EMPTY = new Optional<>();
    private final T value;
    …
}
```

isPresent() checks value of Optional

The Optional class provides the following static methods which are used to create Optionals of type T:

```
static <T> Optional<T> of(T value);
static <T> Optional<T> ofNullable(T value);
static <T> Optional<T> empty();
```

since Java 8

Creating an Optional

```
public static void main(String[] args) {
    Optional<String> o1 = null;
    try {
        o1 = Optional.of(null);
    } catch (NullPointerException e) {
        System.out.println(o1 + " - NullPointerException");  //null –
                                                             //NullPointerException
    }
    Optional<String> o2 = Optional.of("Hello");
    System.out.println(o2 + " - OK");                        //Optional[Hello] - OK
    Optional<String> o3 = Optional.ofNullable(null);
    System.out.println(o3 + " - OK");                        //Optional.empty - OK
    Optional<String> o4 = Optional.ofNullable("Hello");
    System.out.println(o4+ " - OK");                         //Optional[Hello] - OK
    Optional<String> o5 = Optional.empty();
    System.out.println(o5);                                  //Optional.empty
}
```

Determining If an Optional Is Present

The **isPresent** method returns **true** if the Optional contains a non-null object and returns **false** otherwise.

```
public boolean isPresent() {
    return value != null;
}
```

In Java 11, the **isEmpty** method was added. This method returns **true** if the Optional wraps a null object and **false** otherwise. This method gives the opposite result of isPresent.

```
public boolean isEmpty() {
    return value == null;
}
```

Determining If an Optional Is Present

```
public static void main(String[] args) {
    Optional<String> o5 = Optional.empty();
    if (o5.isPresent()) {
        System.out.println("o5 is non-null");
    } else {
        System.out.println("o5 is null");            //o5 is null
    }

    Optional<String> imNull = Optional.ofNullable(null);
    if (imNull.isEmpty()) {
        System.out.println("Empty");                 //Empty
    }
}
```

Retrieving the Contents of an Optional

The **get** method returns the object wrapped by an Optional.

```
public T get() {
    if (value == null) {
        throw new NoSuchElementException("No value present");
    }
    return value;
}
public static void main(String[] args) {
    Optional<String> o4 = Optional.ofNullable("Hello");
    Optional<String> o5 = Optional.empty();
    try {
        o5.get();
    } catch (NoSuchElementException e) {
        System.out.println("NoSuchElementException");
                                        //NoSuchElementException
    }
    …
```

Retrieving the Contents of an Optional

```
    …
    /*Always check ifPresent before calling get*/
    if (o4.isPresent()) {
        System.out.println(o4.get());          //Hello
    }
    if (o5.isPresent()) {
        System.out.println(o5.get());
    } else {
        System.out.println("o5 is null");      //o5 is null
    }
    if (!o5.isEmpty()) {
        System.out.println(o5.get());
    } else {
        System.out.println("o5 is null");      //o5 is null
    }
}
```

Creating Chains of Optionals

The Optional class provides the following methods that can be used to create chains of Optionals:

T **orElse**(T other);

T **orElseGet**(Supplier<? extends T> supplier);

Optional<T> **or**(Supplier<? extends Optional<? extends T> > supplier);

<X extends Throwable> T **orElseThrow**(Supplier<? extends X> exceptionSupplier);

T **orElseThrow()**; [JAVA10]

```
public static void main(String[] args) {
    String t = null;
    String u = "Hello";
    /*Program needs to pick the string that is non-null*/
    String s = t;
    if (t == null) {
        s = u;
    }   …
```

Creating Chains of Optionals

```
…
/*Using Optional method to pick the string that is non-null*/
s = Optional.ofNullable(t).orElse(u);
/*The orElseGet method accepts a supplier which returns the
  Optional value, otherwise returns the result produced by the
  supplying function.*/
String s2 = Optional.ofNullable(t).orElseGet(() -> u);
/*The orElseThrow method accepts a supplier which provides an
  exception which is thrown if the value is not present*/
try {
    s = null;
    String opt = Optional.ofNullable(s).orElseThrow(()
            -> new Exception("Null Optional"));
} catch (Exception e) {
    System.out.println(e.getMessage());          //Null Optional
}
…
```

Creating Chains of Optionals

```
…
/*In Java 10, an overloaded version of the orElseThrow method was
   provided. This version has no arguments and throws a
   NoSuchElementException*/
try {
    s = null;
    String opt = Optional.ofNullable(s).orElseThrow();
} catch (Exception e) {
    System.out.println(e.getMessage());     //No value present
}
/*The orElse, orElseGet, and orElseThrow methods return a value of
   type X. The or method returns an Optional<X> that can be followed
   by additional links in an Optional chain*/
Supplier<Optional<String>> supplier = () -> {
    System.out.print("Enter a string:");
    return Optional.of((new Scanner(System.in)).nextLine());
};
…
```

Creating Chains of Optionals

```
    …
    s = null;
    Optional<String> os = Optional.ofNullable(s).or(supplier);
    if (os.isPresent()) {
        System.out.println(os.get());
    }
}
```

Printing the Contents of an Optional

The Optional class's **ifPresent** method accepts a consumer and is frequently used to print the object that is wrapped by an Optional.

```
void ifPresent(Consumer<? super T> action);
```

Since ifPresent returns void, it must be the last method called if used in an Optional chain.

```
public static void main(String[] args) {
    Supplier<Optional<String>> supplier = () -> {
        System.out.print("Enter a string:");
        return Optional.of((new Scanner(System.in)).nextLine());
    };
    String s = null;
    Optional.ofNullable(s).or(supplier).ifPresent(x ->
                                        System.out.println(x));
    }
```

Filtering Optionals

The Optional class's **filter** method accepts a predicate and returns an Optional, so it can be used anywhere within an Optional chain.

```
Optional<T> filter(Predicate<? super T> predicate);
```

The filter method returns the Optional if the predicate is true and returns an empty Optional if the predicate is false.

```
public static void main(String[] args) {
    String t = null;
    Optional<String> op = Optional.ofNullable(t).filter(x -> x.length() > 2);
    System.out.println(op);                                    //Optional.empty
    Optional.of("Hello")
            .filter(x -> x.startsWith("H"))
            .filter(x -> x.length() > 2)
            .filter(x -> x.charAt(1) == 'e')
            .ifPresent(x -> System.out.println(x));        //Hello
    …
```

Filtering Optionals

```
…
/*The first predicate is false, so the second and third predicates
  do not execute and no output is produced*/
Optional.of("Hello")
        .filter(x -> x.startsWith("i"))
        .filter(x -> x.length() > 2)
        .filter(x -> x.charAt(1) == 'e')
        .ifPresent(x -> System.out.println(x));     //print nothing
/*Filter conditions can be logically OR'ed using the Predicate class's
  or method along a predicate chain*/
Predicate<String> p = x -> x.charAt(0) == 'i';
Optional.of("Hello")
        .filter(p.or(x -> x.startsWith("H")))
        .ifPresent(x -> System.out.println(x));     //Hello
}
```

Optional Chains Involving map and flatmap

The Optional class's **map** method accepts a function that converts a non-null Optional<X> to an Optional<Y>.

<U> Optional<U> **map**(Function<? super T, ? extends U> mapper);

```
public static void main(String[] args) {
    Optional.of("4")
            .map(x -> Integer.parseInt(x))
            .filter(x -> x > 2)
            .filter(x -> x % 2 == 0)
            .ifPresent(x -> System.out.println(x));          //4
    /*Since the ifPresent method returns void, any modifications
      to the underlying object will not persist*/
    Optional<Integer> o1 = Optional.of(2);
    o1.ifPresent(x -> ++x);
    o1.ifPresent(x -> System.out.println(x));                //2
    …
```

Optional Chains Involving map and flatmap

```
…
/*The map method can be used to modify the underlying object,
   since it returns a new Optional. Simply use the same input and
   output type in the function (by providing an implementation of
   UnaryOperator)*/
Optional.of(2)
        .map(x -> ++x)
        .ifPresent(x -> System.out.println(x));                    //3
/*The flatMap method is similar to the map method, except that the
  result of its function is already wrapped in an Optional*/
Optional.of("4")
        .flatMap(x -> Optional.of(Integer.parseInt(x)))
        .ifPresent(x -> System.out.println(x));                    //4
…
```

```
<U> Optional<U> flatMap(Function<? super T,
                        ? extends Optional<? extends U> > mapper);
```

Optional Chains Involving map and flatmap

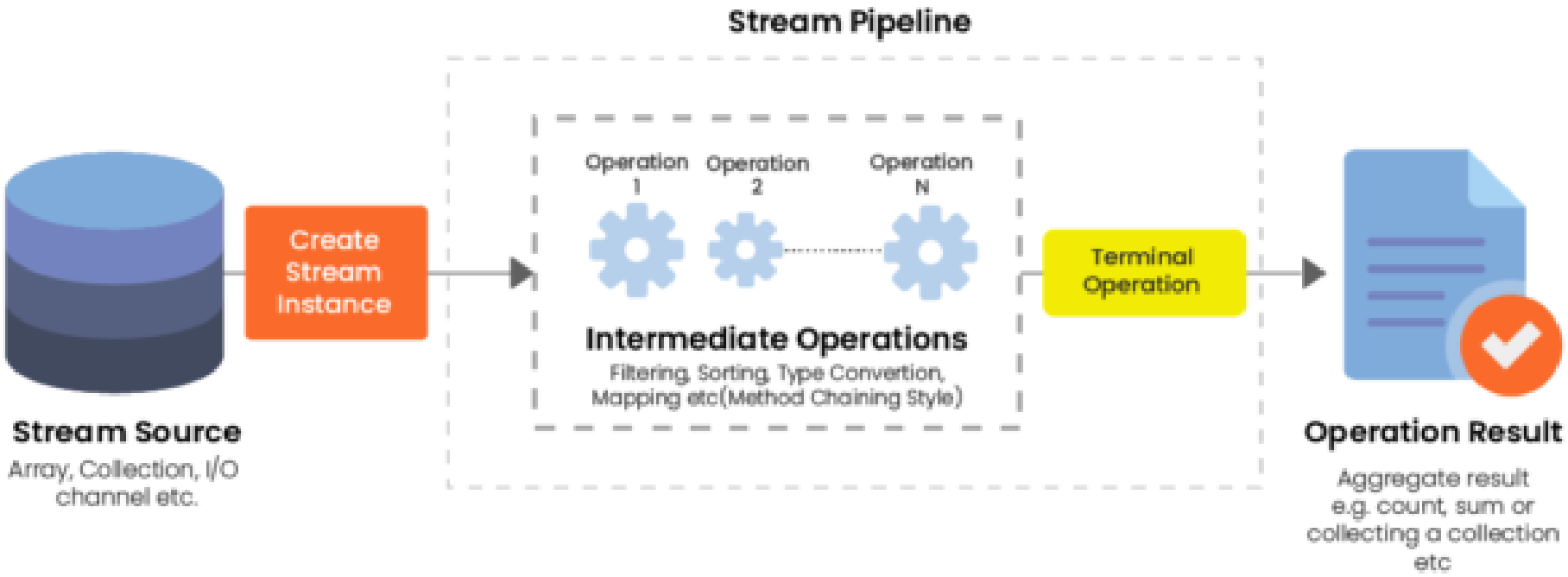
```
…
/*An Optional chain can be used to track consumption of a resource*/
Optional.of(new Resource())          //Resource.count=2
        .filter(x -> x.count > 0)
        .map(x -> new Resource())    //Resource.count=1
        .filter(x -> x.count > 0)
        .map(x -> new Resource())    //Resource.count=0
        .filter(x -> x.count > 0)    //Optional(null)
        .map(x -> new Resource());   //does not execute
}
```

```
class Resource {
    static int count = 3;
    public Resource() {
        count--;
        System.out.println("Resource consumed,
                                + count + " remaining.");  }  }
```

Module contents

1. Functional Programming in Java
   - Functional Interfaces
   - Lambda Expressions
   - Predicates
   - Functions
   - Operators
   - Consumers
   - Suppliers
   - Use in Traversing Objects
   - Use in Collections
   - Use in Comparing Objects
   - Use in Optionals
   - Use in Streams

Java Streams



Interface **java.util.stream.Stream<T>**

Methods:

- Intermediate
- Terminal

since Java 8

Generating Stream Elements

The **Stream** interface provides chainable operations that can be performed on a series of values. A Stream is generic for type parameter T which is the type of its values.

public interface Stream<T> extends BaseStream<T, Stream<T> > {…}

```
public static void main(String[] args) {
    /**Create empty sequential stream*/
        Stream<Integer> nums = Stream.empty();
        nums.forEach(x ->System.out.print("Empty stream: " + x));
                                                    //print nothing
        /*Create sequential ordered stream from values*/
        Stream<Integer> numbers = Stream.of(1, 2, 3, 4);
        numbers.forEach(x ->System.out.print(x + " "));   //1 2 3 4
        System.out.println();
…
```

Generating Stream Elements

```
…
    /* Create a sequential ordered stream according to the initial
       value and the UnaryOperator */
    Stream<Integer> tenIterateNumbers = Stream.iterate(1, x ->
                                                    2 * x).limit(10);
    tenIterateNumbers.forEach(x ->System.out.print(x + " "));
                                        //1 2 4 8 16 32 64 128 256 512
    System.out.println();
    /*Stream generation according to the argument-Supplier */
    Stream<Integer> tenRandomNumbers = Stream.generate(()  ->
       (new Random()).nextInt(100)).limit(10);
    tenRandomNumbers.forEach(x ->System.out.print(x + " "));
                                        //61 21 25 41 87 70 30 78 28 96
    System.out.println();
    /*Генерування стріма за допомогою Builder*/
    Stream.builder().add("First").add("Second").build()
            .forEach(System.out::println);              //First Second
}
```

Converting an Object to a Stream

Stream can be generated from objects:

```
public static void main(String[] args) {
    List<String> list = Arrays.asList("RED", "GREEN");
    list.stream()
          .forEach(x -> System.out.println(x));       //GREEN
    String[] arr = {"RED", "GREEN"};
    Arrays.stream(arr)
           .forEach(x -> System.out.println(x ));     //RED GREEN
    Optional.of("RED")
        .stream()
        .forEach(x -> System.out.println(x));         //RED
    String s = "Java supports functional programming";
    s.lines()
          .forEach(x -> System.out.print(x));   // Java supports functional
                                                //programming
```

…

Converting an Object to a Stream

```
…
    Files.walk(Paths.get("E:\\1"))
            .filter(Files::isRegularFile)
            .filter(p->p.toString().endsWith("jpg"))
            .forEach(System.out::println);          //list of filname

    String fileCOntent =Files.lines(Paths.get("c:\\fil.txt"))
                .collect(Collectors.joining(","));
    System.out.println(fileCOntent);                //comma separated
}                                                   //file content lines
```

Traversing Streams

Methods of Sream interface are ***intermediate*** or ***terminal***. Collection objects can be traversed by providing a consumer to the object's **forEach** method. Streams can be traversed in the same manner. **forEach is *terminal* method.**

```
public static void main(String[] args) {
    Stream<Integer> tenRandomNumbers = Stream.generate(()
            -> (new Random()).nextInt(100))
            .limit(10);
    /*public abstract void forEach(Consumer<? super T> action)*/
    tenRandomNumbers.forEach(x -> System.out.println(x));
}
```

- **Stream<T> object can not be used more than once;**
- **Stream<T> data processing will begin when the terminal method is invoked.**

Filtering Stream Elements

The **filter** method removes elements from the stream that do not match its predicate (**filter** is ***intermediate*** method).

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

```
public static void main(String[] args) {
    Stream.of("RED", "GREEN", "BLUE", "RED")
            .filter(x -> x.equals("YELLOW"))
            .forEach(x -> System.out.println(x));                    //print nothing
    Stream.of("RED", "GREEN", "BLUE", "RED")
            .filter(x -> x.equals("RED"))
            .forEach(x -> System.out.print(x + " "));          //RED RED
    System.out.println();
    Predicate<String> isRed = x -> x.equals("RED");
    Stream.of("RED", "GREEN", "BLUE", "RED")
            .filter(isRed.or(x -> x.indexOf("R") > -1))
            .forEach(x -> System.out.print(x + " "));     //RED GREEN RED
}
```

Sorting Stream Elements

The following ***intermediate*** methods can be used to sort the elements in a stream:

```
Stream<T> sorted();                          //sorted in natural order
Stream<T> sorted(Comparator<? super T> comparator);

public static void main(String[] args) {
    Stream.of("Kyle", "Jaquiline", "Jimmy")
            .sorted()
            .forEach(x -> System.out.print(x + " ")); //Jaquiline Jimmy Kyle
    System.out.println();
    Stream.of("Kyle", "Jaquiline", "Jimmy")
            /*Comparator using*/
            .sorted((x, y)
                -> removeVowels(x).compareTo(removeVowels(y)))
            .forEach(x -> System.out.print(x + " ")); //Jimmy Jaquiline Kyle
    System.out.println();
}
```

Selecting the Smallest or Largest Element in a Stream

The **min** method selects the smallest element in the stream according to the comparator provided in its argument. The **max** method selects the largest element in the stream according to the comparator provided in its argument. Both methods return an Optional of the same type as the stream elements and are ***terminal*** methods.

```
Optional<T> min(Comparator<? super T> comparator);
Optional<T> max(Comparator<? super T> comparator);
```

Selecting the Smallest or Largest Element in a Stream

```
public class SmallestLargestElementSelection {
    private static String removeVowels(String s) {
        return s.replaceAll("[aeiou]", "");
    }
    public static void main(String[] args) {
        Stream.of("Kyle", "Jaquiline", "Jimmy") // Stream<String>
            .min((x, y) -> removeVowels(x).compareTo(removeVowels(y)))
                                                    //Optional<String>
            .ifPresent(x -> System.out.println(x));       //Jimmy

        Stream.of("Kyle", "Jaquiline", "Jimmy") // Stream<String>
            .max((x, y) ->
removeVowels(x).compareTo(removeVowels(y)))      //Optional<String>
            .ifPresent(x -> System.out.println(x));       //Kyle
    }
}
```

flatMap vs map

The **map** method does not perform flattening. For a transformation of a Stream of type T to a Stream of type R, the map method accepts a function whose second type parameter is R.

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
…
        Stream.of(
            new ClassForSubject("Biology",
                    new Student("Joe", 1001, 3.81),
                    new Student("Mary", 1002, 3.91)),
            new ClassForSubject("Physics",
                    new Student("Kalpana", 1003, 3.61),
                    new Student("Javier", 1004, 3.71)))     // Stream<ClassForSubject>
            .map(x -> x.students)                           //Collection<Student>>
           .forEach(x -> System.out.println(x));            //Print distributed by class
                                                            // students
}
```

Both flatMap and map methods are intermediate.

Output:

[Joe 1001 3.81, Mary 1002 3.91]
[Kalpana 1003 3.61, Javier 1004 3.71]

flatMap vs map

The **flatMap** method flattens a stream during the transformation of elements to a target type. For a transformation of a Stream of type T to a Stream of type R, the flatMap method accepts a function whose second type parameter is Stream<? extends R>.

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

```
public class Student {
   String name;
   Integer id;
   Double gpa;
   public Student(String n, int i, double g)
{      name = n;
       id = i;
       gpa = g;         }
   @Override
   public String toString() {
       return name + " " + id + " " + gpa;
}  }
```

```
public class ClassForSubject {
    String subject;
    Collection<Student> students;
    public ClassForSubject(String su,
                                Student... st) {
    subject = su;
        students = Arrays.asList(st);
    }
}
```

flatMap vs map

```
public static void main(String[] args) {
    Stream.of(new ClassForSubject("Biology",
            new Student("Joe", 1001, 3.81),
            new Student("Mary", 1002, 3.91)),
            new ClassForSubject("Physics",
                    new Student("Kalpana", 1003, 3.61),
                    new Student("Javier", 1004, 3.71)))
                                                    // Stream<ClassForSubject>
            .flatMap(x -> x.students.stream())      // Stream<Student>
            .forEach(x -> System.out.println(x));   //Print all 4 students
                                                    //of both classes
```

...

The Function converts the Class's student collection to a stream which is flattened by flatMap to create a stream whose elements are the students in both classes.

Output:

Joe 1001 3.81
Mary 1002 3.91
Kalpana 1003 3.61
Javier 1004 3.71

Reducing Stream Elements

The **reduce** method applies a binary operation to each element of the stream to produce a reduction in the form of an Optional object.

```
public abstract Optional<T> reduce(BinaryOperator<T> accumulator);
```

```
Stream.of(1, 2, 3, 4, 5)                          //Stream(Integer)
        .reduce((x, y) -> x * y)                   //Optional(Integer)
        .ifPresent(x -> System.out.println(x));  //Prints 120
```

The **reduce** method has an overloaded version which requires an identity, which is the initial value for the running computation. It returns a T, which is the type of the stream elements.

```
public abstract T reduce(T identity, BinaryOperator<T> accumulator);
```

```
Stream.of(1, 2, 3, 4, 5)                          //Stream(Integer)
        .reduce(2, (x, y) -> x * y)                //Optional(Integer)
        .ifPresent(x -> System.out.println(x));  //Prints 240
```

Reducing Stream Elements

The **reduce** method has another overloaded version which can transform complex objects into simpler reductions.

```
public abstract <U> U reduce(U identity, BiFunction<U,? super T,U>
                                    accumulator, BinaryOperator<U> combiner)
```

```
/*Program needs to sum all second integers in a Stream of TwoInts*/
Stream<TwoInts> two = Stream.of(new TwoInts(1,2),new TwoInts(8,9));
BiFunction<Integer,TwoInts,Integer> accumulator = (x,y) -> x + y.i2;
BinaryOperator<Integer> combiner = (x,y) -> x += y;
Integer j = two.reduce(0,accumulator,combiner);
System.out.println(j);                                    //11
```

```
public class TwoInts {
    Integer i1;
    Integer i2;
    public TwoInts(int i1, int i2) {
        this.i1 = i1;
        this.i2 = i2;
    } }
```

Reducing Stream Elements

```
public static void main(String[] args) {
   List<Integer> numbers = Arrays.asList(1, 2, 3, 5, 7);
   /*Search min value*/
   Integer min = numbers.stream()
         .reduce(Integer.MAX_VALUE, (left, right) -> left < right ? left : right);
   System.out.println(min);                    //1
   /* Search max value (Integer::max === (a, b) -> Integer.max(a, b))*/
   Integer max = numbers.stream()
         .reduce(Integer.MIN_VALUE, Integer::max);
    System.out.println(max);                   //7
   /*Search longest string*/
   List<String> strings = Arrays.asList("aaa", "bbb", "ccc", "ddd", "ffff");
    String s = strings.stream()
         .reduce("", (left, right) -> left.length() > right.length() ? left : right);
    System.out.println(s);                     //ffff
}
```

Integer.MAX_VALUE – identity ,

left, right) -> left < right ? left : right - accunulator

Collecting Stream Elements into a Mutable Reduction

- A ***collector*** is a general construct for generating *composite values* from streams. The collector can be used with an arbitrary stream by passing it as an argument to the **`collect`** method.
- With the help of collectors, you can collect all the elements in a list, set or other collection, group elements by some criterion, combine everything into a string, etc.
- There are a number of useful pre-built collectors in the standard library.
- The **java.util.stream.Collectors** class has a lot of methods for all occasions, we'll look at them later.
- Optionally, you can write your own collector by implementing the **java.util.stream.Collector** interface - is an abstraction of a reduction operation that accumulates input elements into a mutable container, converting the accumulated result into a final representation after processing all input elements.

Collecting Stream Elements into a Mutable Reduction

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
    public static<T, R> Collector<T, R, R> of(Supplier<R> supplier,
                                              BiConsumer<R, T> accumulator,
                                              BinaryOperator<R> combiner,
                                              Characteristics... characteristics) {…}
    public static<T, A, R> Collector<T, A, R> of(Supplier<A> supplier,
                                                 BiConsumer<A, T> accumulator,
                                                 BinaryOperator<A> combiner,
                                                 Function<A, R> finisher,
                                                 Characteristics... characteristics) {…}
    enum Characteristics {…}    }
```

Collecting Stream Elements into a Mutable Reduction

The **Collector<T, A, R>** interface has three type variables:

- **T** - type of input elements of the reduction operation;
- **A** - type of variable accumulation of the reduction operation (often hidden as a detail of the implementation);
- **R** - type of result of the reduction operation.

The collector is defined by four functions that work together to accumulate records in a mutable result container, and perform the final transformation of the result:

- **Supplier <A> supplier()** - a function that creates and returns a mutable result container (supplier);
- **BiConsumer <A, T> accumulator ()** - a function that performs the addition of a new data element to the results container (drive);
- **BinaryOperator <A> combiner ()** - a function that combines two partial results into one (combiner);
- **Function <A, R> finisher ()** - a function that performs the optional final conversion of the intermediate data type of the battery to the final data type of the result of the operation (finisher).

Collecting Stream Elements into a Mutable Reduction

The Collector interface also contains a set of characteristics in the Characteristics enumerator, which provide hints on how implementations of the reduction operation functions can be used:

- CONCURRENT - indicates that the battery function can be called by several execution streams simultaneously;
- UNORDERED - indicates that the reduction operation does not guarantee the order of the elements equal to their input order;
- IDENTITY_FINISH - indicates that the finisher finisher function () is an identity function and can be removed.

A sequential implementation of the collector reduction will create a single result container using the vendor function, and call the drive function once for each input element. A parallel implementation will split the input, create a results container for each part, accumulate the contents of each part in the sub-result for that partition, and then use the combiner function to combine the sub-results into a combined result.

Collecting Stream Elements into a Mutable Reduction

Suppose a program needs to arrange the elements of a character stream such that alphabetic characters appear before numeric characters. A mutable reduction of the character stream can be created that is arranged in this order. This reduction is generated using an instance of the **Collector** interface.

```
public static void main(String[] args) {
    Supplier<List<Character>> supp = () -> new ArrayList<Character>();
    BiConsumer<List<Character>, Character> acc = (x, y) -> {
        System.out.print("acc: x=" + x + " y=" + y + " result=");
        if (Character.isAlphabetic(y)) {
            x.add(0, y);
        } else {
            x.add(y);
        }
        x.forEach(z -> System.out.print(z));
        System.out.println();
    };
```

Collecting Stream Elements into a Mutable Reduction

```
…
   BinaryOperator<List<Character>> comb1 = (x, y) -> {
      x.addAll(y);
      return x;
   };
   Stream.of('1', 'a', 'b', '2')                          // Stream<Character>
      .collect(Collector.of(supp, acc, comb1))            //List<Character>
      .forEach(x -> System.out.print(x + " "));
   System.out.println();
```

Output:

```
acc: x=[] y=1 result=1
acc: x=[1] y=a result=a1
acc: x=[a, 1] y=b result=ba1
acc: x=[b, a, 1] y=2 result=ba12
b a 1 2
```

Collecting Stream Elements into a Mutable Reduction

```
…
BiConsumer<List<Character>, List<Character>> comb2 = (x, y)
        -> x.addAll(y);
Stream.of('1', 'a', 'b', '2')
        .collect(supp, acc, comb2)
        .forEach(x -> System.out.print(x + " "));
System.out.println();

Supplier<StringBuilder> supps = ()
        -> new StringBuilder();
BiConsumer<StringBuilder, String> accs = (x, y)
        -> x.append(y.replaceAll("[aeiou]", ""));
BinaryOperator<StringBuilder> combs = (x, y) -> {
    x.append(y);
    return x;
};
Function<StringBuilder, String> fins = x -> x.toString();
String s = Stream.of("Joe", "Kalpana", "Christopher")
        .collect(Collector.of(supps, accs, combs, fins));
System.out.println(s);    }
```

Output:

```
acc: x=[] y=1 result=1
acc: x=[1] y=a result=a1
acc: x=[a, 1] y=b result=ba1
acc: x=[b, a, 1] y=2 result=ba12
b a 1 2
```

Output:

```
JKlpnChrstphr
```

Using Prewritten Collectors

The **java.util.stream.Collectors** class contains static methods which create **Collector** objects that solve various problems.

```
/* toList  accumulates the stream elements into a list in the original order
    public static <T>  Collector<T, ?, List<T>> toList() */
List<Character> list = Stream.of('1', 'a', 'b', '2')
        .collect(Collectors.toList());
System.out.println(list);       //[1, a, b, 2]

/*public static <T> Collector<T, ?, Set<T>> toSet() */
Set<Character> set = Stream.of('1', 'a', '1', 'a')
        .collect(Collectors.toSet());
System.out.println(set);        //[1, a]

/*public static <T, C extends Collection<T>> Collector<T, ?, C>
                                toCollection(Supplier<C> collectionFactory) */
Deque<Integer> deque = Stream.of(1, 2, 3, 4, 5)
                                .collect(Collectors.toCollection(ArrayDeque::new));
System.out.println(deque);      //[1, 2, 3, 4, 5]
```

Using Prewritten Collectors

```
…
 /*public static <T, K, U> Collector<T, ?, Map<K,U>>
      toMap(Function<? super T, ? extends K> keyMapper,
          Function<? super T, ? extends U> valueMapper) */
Map<Integer, String> map1 = Stream.of(1, 2, 3)
    .collect(Collectors.toMap(
        Function.identity(),       //returns current stream element value
        i -> String.format("%d", i * 2)
));
map1.forEach((k, v) -> System.out.print(k + "=" + v + " "));      // 1=2 2=4 3=6
/*The collectingAndThen method collects the elements of a stream
   using a Collector object and then transforms the result to another type
   using a function
   static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(
                        Collector<T,A,R> downstream,
                        Function<R,RR> finisher);*/
Supplier<List<Character>> supp = () -> new ArrayList<Character>();
…
```

Using Prewritten Collectors

```
…
    BiConsumer<List<Character>, Character> accc2 = (x, y) -> {
            if (Character.isAlphabetic(y)) {
                x.add(0, y);
            } else {
                x.add(y);
            }
        };
    BinaryOperator<List<Character>> comb1 = (x, y) -> {
        x.addAll(y);
        return x;
    };
    Function<List<Character>, String> fins2 = x -> {
        String t = "";
        for (Character c : x) {
            t += c;
        }
        return t;
    };
…
```

Using Prewritten Collectors

```
…
   String t = Stream.of('1', 'a', 'b', '2')                    //Stream<Character>
         .collect(Collectors.collectingAndThen(
               Collector.of(supp, accc2, comb1),              //List<Character>
               fins2));                                       //String
   System.out.println(t);                                     //ba12

   /*The reducing method collects the elements of a stream into an Optional
     object. It accepts a BinaryOperator.
     static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> op)*/
   Stream.of(1, 2, 3, 4, 5)                                   // Stream<Integer>
         .collect(Collectors.reducing((x, y)
               -> x *= y))                                    // Optional<Integer>
         .ifPresent(System.out::println);          //120
…
```

Using Prewritten Collectors

```
…
   /*The joining method joins a stream of strings into a single String object.
     static Collector<CharSequence,?,String> joining()*/
   String s = Stream.of("RED", "GREEN", "BLUE")          // Stream<String>
         .collect(Collectors.joining());                 // String
   System.out.println(s);                                //REDGREENBLUE

   /*A delimiter can also be specified*/
   s = Stream.of("RED", "GREEN", "BLUE")                 // Stream<String>
         .collect(Collectors.joining(","));              // String
   System.out.println(s);                                //RED,GREEN,BLUE

…
```

Using Prewritten Collectors

```
...
   /*The groupingBy method organizes the result by a key value. The resulting
     container is a map whose values are a list of stream elements. A function
     which transforms a stream element to a key value must be provided
     static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
                          Function<? super T,? extends K> classifier)*/

   class Car {
       String manu;
       String model;
       int mpg;
       public Car(String ma, String mo, int mp) {
           manu = ma;
           model = mo;
           mpg = mp;
       }
       public String toString() {
           return manu + " " + model + " gets " + mpg + " mpg";
       }
   }
```

Using Prewritten Collectors

```
…
   /*The groupingBy method organizes the result by a key value. The resulting
     container is a map whose values are a list of stream elements. A function
     which transforms a stream element to a key value must be provided
     static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
                         Function<? super T,? extends K> classifier)*/

   Stream.of(new Car("Buick", "Regal", 25),
         new Car("Hyundai", "Elantra", 27),
         new Car("Buick", "Skylark", 26),
         new Car("Hyundai", "Accent", 30))          //Stream<Car>
         .collect(
               Collectors.groupingBy(x -> x.manu))      //Map<String,List<Car>>
      .forEach((x, y) -> System.out.println(x + ": " + y));
   /*Buick: [Buick Regal gets 25 mpg, Buick Skylark gets 26 mpg]
     Hyundai: [Hyundai Elantra gets 27 mpg, Hyundai Accent gets 30 mpg]*/
…
```

Using Prewritten Collectors

```
…
   /*An overloaded version of the groupingBy method accepts a Collector
     object as its second argument. The mapping method is frequently used
     as the second argument to transform the stream element and then generate
     a new list as the mapped value.
     static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,
             ? extends U> mapper, Collector(<? super U,A,R> downstream);*/
   Stream.of(new Car("Buick", "Regal", 25),
         new Car("Hyundai", "Elantra", 27),
         new Car("Buick", "Skylark", 26),
         new Car("Hyundai", "Accent", 30))          //Stream<Car>
         .collect(Collectors.groupingBy(x -> x.manu,   //Map<String,List<Integer>>
               Collectors.mapping(x -> x.mpg, Collectors.toList()))) //List<Integer>>
         .forEach((x, y) -> System.out.println(x + ": " + y));
    /*Buick: [25, 26]
      Hyundai: [27, 30]*/
…
```

Using Prewritten Collectors

```
…
   /*The partitioningBy method splits the reduction into two components
     based on a pass/fail criterion. The map's key is of type Boolean,
     and a predicate is provided to specify the pass/fail criterion.
     static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
                                    Predicate<? super T> predicate)*/
   Stream.of(new Car("Buick", "Regal", 25),
         new Car("Hyundai", "Elantra", 27),
         new Car("Buick", "Skylark", 26),
         new Car("Hyundai", "Accent", 30))                  //Stream<Car>
         .collect(Collectors.partitioningBy(x -> x.mpg >= 30))
                                                            //Map<Boolean,List<Car>>
            .forEach((x, y) -> System.out.println(x + ": " + y));
   /*false: [Buick Regal gets 25 mpg, Hyundai Elantra gets 27 mpg,
                                                  Buick Skylark gets 26 mpg]

     true: [Hyundai Accent gets 30 mpg]*/
…
```

Using Prewritten Collectors

```
...
/*The Collectors class has methods which produce a sum as the mutable
   reduction. These methods accept a function which transforms the stream
   element into the int, long, or double to be summed.
   static <T> Collector<T, ? , Integer> summingInt(ToIntFunction<? super T>
Integer sum                                                    mapper)*/
        = Stream.of(new Car("Buick", "Regal", 25),
              new Car("Hyundai", "Elantra", 27),
              new Car("Buick", "Skylark", 26),
              new Car("Hyundai", "Accent", 30))
              .collect(Collectors.summingInt(x -> x.mpg));
System.out.println("sum of mpg = " + sum);          //sum of mpg = 108
/*Also public static <T> Collector<T, ?, Double> averagingInt(ToIntFunction<?
                                                      super T> mapper)/*/

Double ave
          = Stream.of(new Car("Buick", "Regal", 25),
                new Car("Hyundai", "Elantra", 27),
                new Car("Buick", "Skylark", 26),
                new Car("Hyundai", "Accent", 30))
                .collect(Collectors.averagingInt(x -> x.mpg));
   System.out.println("Average of mpg = " + ave);  //Average of mpg = 27.0
```

Using Prewritten Collectors

```
...
   /*The Collectors class has methods which produce a statistics as the mutable
     reduction. These methods accept a function which transforms the stream
     element of the int, long, or double.
     public static <T> Collector<T, ?, IntSummaryStatistics>
   Object stat                    summarizingInt(ToIntFunction<? super T> mapper) */
         = Stream.of(new Car("Buick", "Regal", 25),
                new Car("Hyundai", "Elantra", 27),
                new Car("Buick", "Skylark", 26),
                new Car("Hyundai", "Accent", 30))
          .collect(Collectors.summarizingInt(x -> x.mpg));
      System.out.println("Statistics: " + stat); //Statistics: IntSummaryStatistics{
                        //count=4, sum=108, min=25, average=27,000000, max=30}
   /*Also there is element counting method
     public static <T> Collector<T, ?, Long> counting()*/
   Long count = Stream.of(new Car("Buick", "Regal", 25),
              new Car("Hyundai", "Elantra", 27),
              new Car("Buick", "Skylark", 26),
              new Car("Hyundai", "Accent", 30))
              .collect(Collectors.counting());
      System.out.println(count);                    //4
```

Building Streams Interactively

The **Stream.Builder** interface is a subinterface of **Consumer**. It can be used to interactively build a stream. The Stream interface's static **builder** method is used to create a Stream.Builder.

```
public static void main(String[] args) {
    Stream.Builder<String> bld = Stream.builder();          //get builder
        bld.accept("RED");
        bld.accept("GREEN");
        bld.accept("BLUE");
        Stream<String> st = bld.build();                    //build immutable stream
        st.forEach(x -> System.out.print(x + " "));         //RED GREEN BLUE
        System.out.println();

        try {
            bld.accept("YELLOW");          //can not add elements to builted stream
        } catch (IllegalStateException e) {
            System.out.println("IllegalStateException");       //IllegalStateException
        }
…
```

Building Streams Interactively

```
…
    /*Since the add method returns a Stream.Builder object, it can be used
      in a stream chain (the accept method returns void so it must be used
      as a separate statement).*/
    Stream.builder()
            .add("RED")
            .add("GREEN")
            .add("BLUE")
            .build()
            .forEach(x -> System.out.print(x + " "));      //RED GREEN BLUE
}
```

Displaying Intermediate Results

The **peek** method of Stream interface is useful for reporting and debugging purposes. It accepts a Consumer object of the same type as the stream elements. Stream<T> peek(Consumer<? super T> action)

```
Stream.of(1, 2, 3, 4)
        .peek(x -> System.out.print(x + " "))          //1 2 3 4
        .reduce((x, y) -> x += y)
        .ifPresent(x -> System.out.println(x));        //10
```

In Java 9 the methods have been added to the Stream interface:

```
default Stream<T> takeWhile(Predicate<? super T> predicate)
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

The first returns the elements of the stream as long as they satisfy the condition. The method works similarly to the Stream <T> limit (long maxSize, but with a condition. The second skips the elements as long as they satisfy the condition, and then returns the rest of the stream. If the predicate returns false for the first element, no element will be skipped. The method is similar to the Stream <T> skip (long n), only it works by condition.

Displaying Intermediate Results

```
Stream.of(1, 2, 3, 4, 2, 5)
      .takeWhile(x -> x < 3)
      .forEach(x -> System.out.print(x + " "));        //1 2
    System.out.println();

Stream.of(1, 2, 3, 4, 2, 5)
      .dropWhile(x -> x < 3)
      .forEach(x -> System.out.print(x + " "));        //3, 4, 2, 5
    System.out.println();
```

Stream Specializations

The Java API provides the non-generic **IntStream**, **LongStream**, and **DoubleStream** interfaces which support stream of **Integers**, **Longs**, and **Doubles**, respectively.

```
public static void main(String[] args) {
   /*The Stream interface's mapToInt method uses its ToIntFunction
     argument to convert a generic Stream object to an IntStream*/
   IntStream ints
         = Stream.of(new Car("Buick", "Regal", 25),
               new Car("Hyundai", "Elantra", 27),
               new Car("Buick", "Skylark", 26),
               new Car("Hyundai", "Accent", 30))          //Stream<Car>
               .mapToInt(x -> x.mpg);                     //IntStream
   /*The max and min methods compute the largest and smallest elements
     in the stream, respectively. The methods return OptionalInt, OptionalLong
     and OptionalDouble objects, respectively*/
   ints.max()
         .ifPresent(x -> System.out.println(x));          //30
…
```

Stream Specializations

The Java API provides the non-generic **IntStream**, **LongStream**, and **DoubleStream** interfaces which support stream of **Integers**, **Longs**, and **Doubles**, respectively.

```
…
    LongStream.of(1, 2, 3, 4)
            .min()
            .ifPresent(x -> System.out.println(x));              //1

    /*The average method computes the average of the stream elements and
      returns an OptionalDouble*/
    DoubleStream.of(1.1, 2.2, 3.3, 4.4)
            .average()
            .ifPresent(x -> System.out.println(x));              //2.75
    /*The sum method computes the sum of the stream elements and returns
      an int.*/
    int sum = IntStream.of(1, 2, 3, 4)
            .sum();
    System.out.println(sum);                                     //10
}
```