
New Features in Java 8

- Lambda expressions
- Functional interfaces
- Streaming support for Collections

Lambda expressions

- Are a block of java code with parameters
- Can be assigned to variables
- Can be executed one or more times
- Can access final variables from surrounding block
- Represent a functional interface
- Can be passed to other methods like data

Passing expressions – Before Java 8

Previously (inner) classes which implement a certain interface were used as method parameters

Example ActionListener

```

    JButton testButton = new JButton("Test Button");
    testButton.addActionListener(new ActionListener(){
        @Override
        public void actionPerformed(ActionEvent ae){
            System.out.println("Click Detected");
        }
    });

```

Example Comparator

```
public interface Comparator {
    int compare(Object o1, Object o2);
    boolean equals(Object o);
}

class LengthComparator implements Comparator<String>
{
    public int compare(String first, String second) {
        return Integer.compare(first.length(),
                               second.length());
    }
    public boolean equals(Object o) {...}
}

Arrays.sort(strings, new LengthComparator());
```

Before Java 8 Java instances were created and then passed to methods.

Each instance must belong to a certain interface which is defined in the parameter section of the receiving method.

Java compiler can check if the instances passed to a method are of the correct type.

Passing expressions - lambdas

Example EventListener

```
JButton testButton = new JButton("Test Button");  
testButton.addActionListener(  
    e -> System.out.println("Click Detected")  
);
```

Example Comparator

Example Comparable:

```
List<String> stringList = new ArrayList<String>();  
stringList.add("John");  
..  
  
Collections.sort(stringList,  
                    (String s1, String s2) -> s2.compareTo(s1));
```

Defining lambda expressions

Lambda syntax:

input arguments \rightarrow body

Examples parameter list

```
(int x) -> x+1
```

Parameter list with explicit type

```
int x -> x+1
```

Wrong: Parameter type need braces

```
(x)->x+1
```

Type of parameter is deduced by compiler

```
x -> x+1
```

```
(int x,int y) -> x+y
```

```
int x,int y -> x+y
```

Wrong: Parameter type need braces

```
(x,int y) -> x+y
```

Wrong: Do not use parameter with and without type.

```
() -> 42
```

Empty parameter list is fine

Defining lamda expressions

Examples lambda body

```
() -> System.gc()
```

Only one expression

```
(String s1, String s2) -> {  
    return s1.compareTo(s2);  
}
```

return statement. Body needs curly braces

```
(String s1, String s2) -> {  
    if (s1.length() < s2.length())  
        return -1;  
    else if (s1.length() > s2.length())  
        return 1;  
    else  
        return 0;  
}
```

Multi line statement. Body needs curly braces

Lambda type identification

Lambda expressions have a type.

Type of expression is an instance of a functional interface.

The interface is deduced from the context of the expression.

Predefined functional interfaces are defined in package `java.util.functions`.

Lambda expression must have the same parameter types and the same return type of a functional interface.

Examples

Lambda

$x \rightarrow x^2$

One argument. Produces a value with the same type of x .

Lambda

$x \rightarrow \{\text{return } x < 2;\}$

One argument. Produces a boolean value.

Predefined interfaces

Name	Method	Parameters	Description
Supplier<T>	T get()	None	Provides a value of type T. To retrieve the value call get()
Consumer<T>	accept(T t)	One	Acts upon a value but does not return a value
Predicate<T>	boolean test(T t)	One	Represents a boolean function
Function<T,R>	R apply(T t)	One	Take an argument of type T and returns a value of type R
BiConsumer<T,U>	accept(T t, U u)	Two	Accepts two arguments and returns no value
BiFunction<T,U,R>	R apply(T t,U u)	Two	Function that takes two arguments and returns a value of type R

More interfaces in `java.util.functions`

Create your own interfaces

Create an interface with a **single abstract method**. (SAM is another name for functional interfaces.)

```
public interface DumpPrinter {  
    public void doIt();  
}
```

Usage:

```
DumpPrinter dp = () ->  
    System.out.println("I am dumb");  
  
dp.doIt();
```

Scope of lambda expressions

Instance and static variables can be used in the body of a lambda expression

```
class Bar {  
    int i;  
    Foo foo = i -> i * 2;  
};
```

Parameter `i` in `foo` shadows instance variable `i`.

Scope of lambda expressions

Local variables must be final or effectively final when used in the body of a lambda expression:

```
void bar() {  
    int i;  
    Foo foo = i -> i * 2;  
};
```

Wrong:

```
void bar() {  
    int i;  
    Foo foo = i -> i * 2;  
    i = 2;  
};
```

Method references

Lambda expressions are anonymous representations of functional interfaces.

Method references are concrete implementations of an interface by a class which match the required functional interface.

Usage:

`<ClassName>::<methodName>`

`String::valueOf`

`Integer::compare`

Method references

`java.util.Arrays` has a static method

```
public static <T>  
void sort(T[] a, Comparator <T> c);
```

`Integer::compare` has a compatible signature to the `Comparator` interface.

```
Arrays.sort(myIntArray, Integer::compare)
```

Same as:

```
Arrays.sort(myIntArray, (i1, i2) -> {  
    return i1.compareTo(i2);  
});
```

Streams

Streams are an addition to Javas Collections

Streams are a sequence of values

Streams can be processed by lambda expressions

Streams are not a datastructure

Streams, like iterators are consumable. To revisit a stream ask collection for its stream

Streams

A stream is a pipeline of functions.

Streams can transform data.

Streams cannot mutate data.

Think of Streams as Java pipes.

You can create a Collection or an Array from a Stream.

Getting a stream

Ask a collection with `stream()` or `parallelStream()` method.

With `Arrays.stream(Object[])`

Ask a `BufferedReader` instance with `lines()`

Get files in a directory with `Files.list()`

Creating a pipeline

Create a stream from a source

Append intermediate operations like `filter()` or `map()`. Each intermediate operation creates a new stream holding only elements matching the predicate of the intermediate operation.

Append terminal operation to produce a result. After the terminal operation the stream can no longer be used. No processing is done before terminal operation.

Intermediate operations

Function	Description
map()	Returns a stream consisting of the result applying the given function
filter()	Returns a stream consisting of the element that match the given predicate
distinct()	Returns a stream consisting of distinct elements
sorted()	Returns sorted elements
peek()	Applies a consumer to each element. Can be used for debugging.
flatMap()	Returns a stream consisting of the data of streams produced by the given function.

Most useful operations. For all operations see Streams javadoc

Terminal operations

Function	Description
<code>reduce()</code>	Performs a reduction on the elements of this stream to a single value.
<code>collect()</code>	Groups the elements in this stream. You can use the Collectors from package <code>java.util.streams.Collectors</code> .
<code>forEach()</code>	Performs an action for each element of this list.

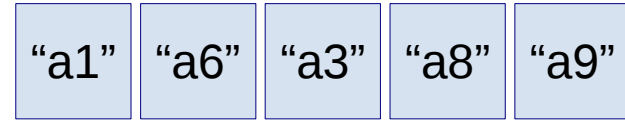
Most useful operations. For all operations see Streams javadoc

Streams revisited

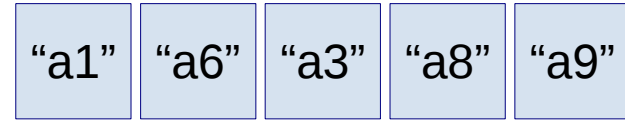
```
public static void pipe7Test() {  
    String sa[] = {"a1", "a6", "a3", "a8", "a9"};  
  
    int summe = Arrays.stream(sa)  
        .map(s -> s.substring(1))  
        .map(s -> Integer.parseInt(s))  
        .filter(i -> i > 6)  
        .reduce(0, Integer::sum);  
  
    System.out.println("Summe: " + summe);  
}
```



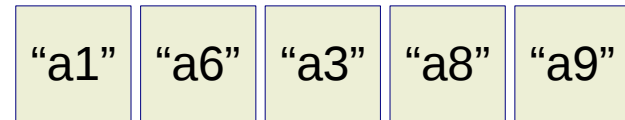
```
String sa[] = {"a1", "a6", "a3", "a8", "a9"};
```



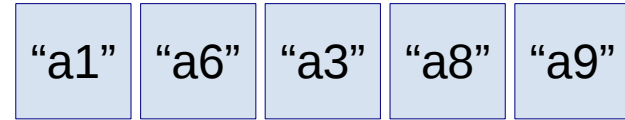
```
String sa[] = {"a1", "a6", "a3", "a8", "a9"};
```



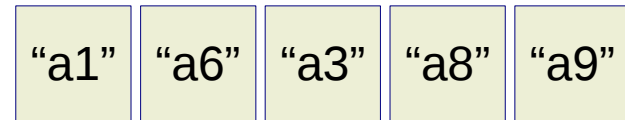
```
Arrays.stream(sa)
```



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

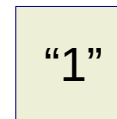


Arrays.stream(sa)

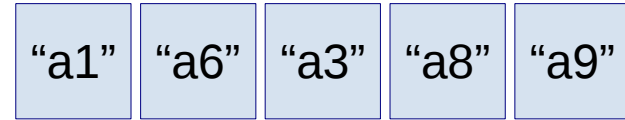


map(s -> s.substring(1))

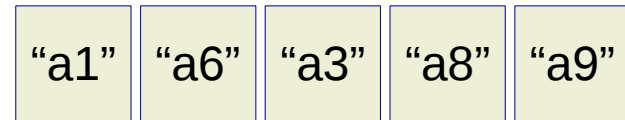
"a1"->"a1".substring(1)



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

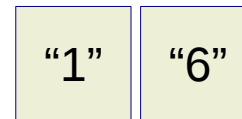


Arrays.stream(sa)

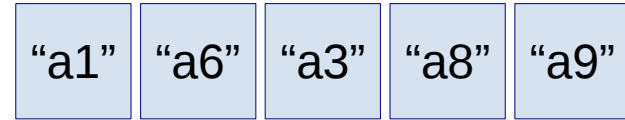


map(s -> s.substring(1))

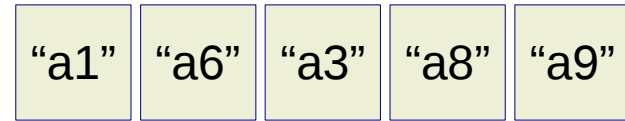
"a6"->"a6".substring(1)



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

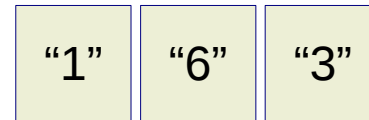


Arrays.stream(sa)

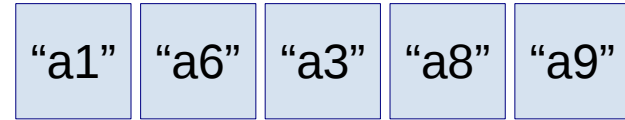


map(s -> s.substring(1))

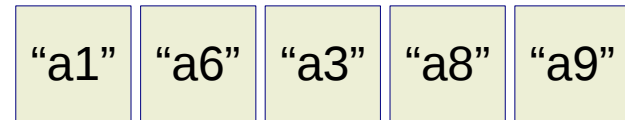
"a3"->"a3".substring(1)



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

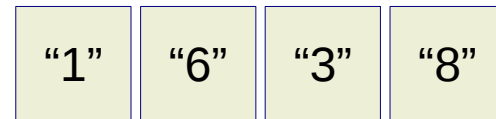


Arrays.stream(sa)

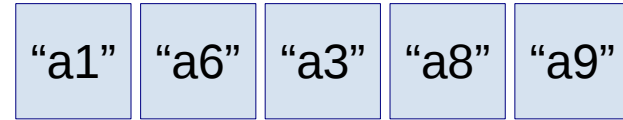


map(s -> s.substring(1))

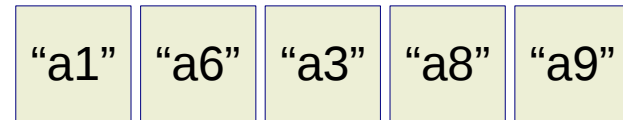
"a8"->"a8".substring(1)



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

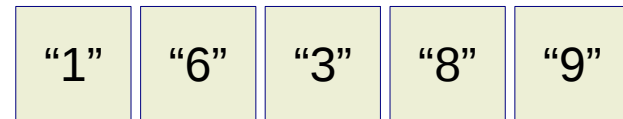


Arrays.stream(sa)



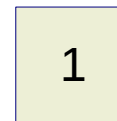
map(s -> s.substring(1))

"a9"->"a9".substring(1)

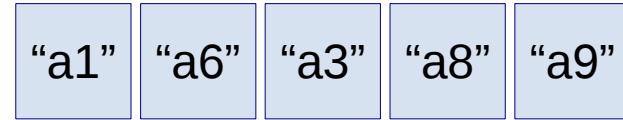


map(s -> Integer.parseInt(s))

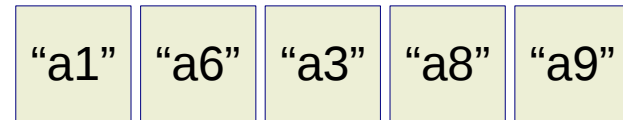
"1"->Integer.parseInt("1")



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

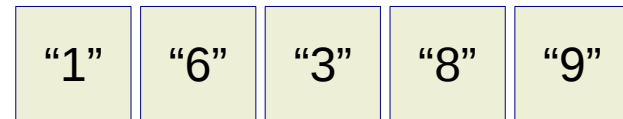


Arrays.stream(sa)



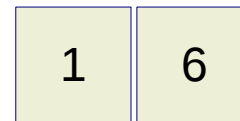
map(s -> s.substring(1))

"a9"->"a9".substring(1)

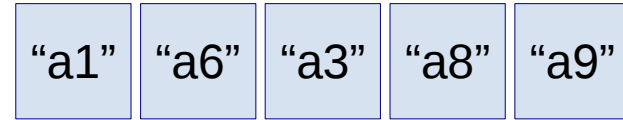


map(s -> Integer.parseInt(s))

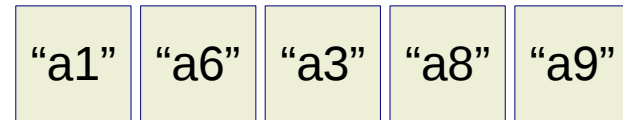
"6"->Integer.parseInt("6")



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

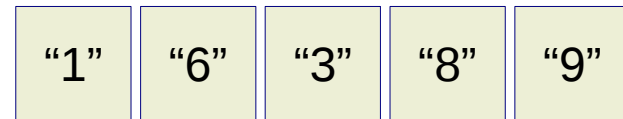


Arrays.stream(sa)



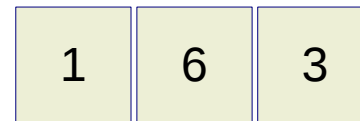
map(s -> s.substring(1))

"a9"->"a9".substring(1)

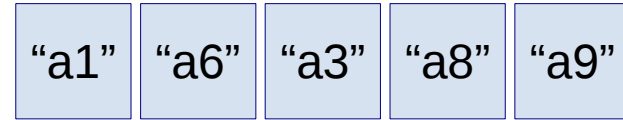


map(s -> Integer.parseInt(s))

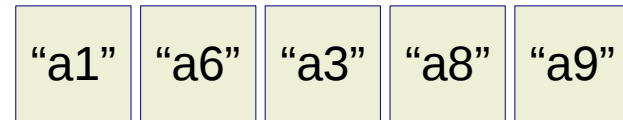
"3"->Integer.parseInt("3")



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

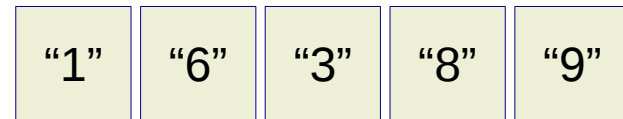


Arrays.stream(sa)



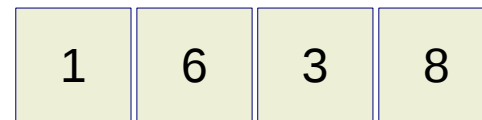
map(s -> s.substring(1))

"a9"->"a9".substring(1)

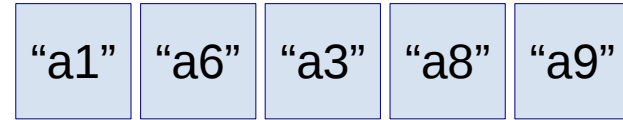


map(s -> Integer.parseInt(s))

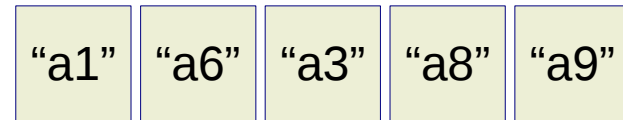
"8"->Integer.parseInt("8")



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

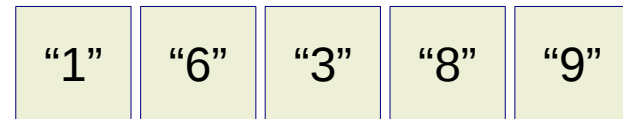


Arrays.stream(sa)



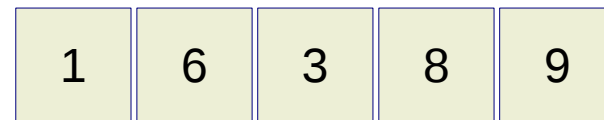
map(s -> s.substring(1))

"a9"->"a9".substring(1)



map(s -> Integer.parseInt(s))

"9"->Integer.parseInt("9")

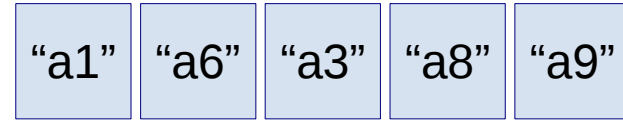


filter(i -> i > 6)

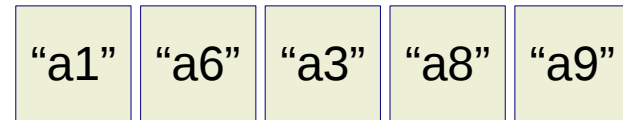
1->1>6

Results false. Nothing is added to result set

String sa[] = {"a1", "a6", "a3", "a8", "a9"};

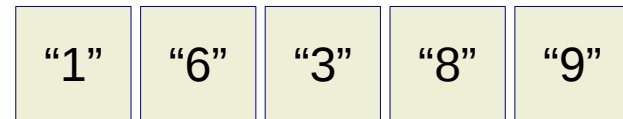


Arrays.stream(sa)



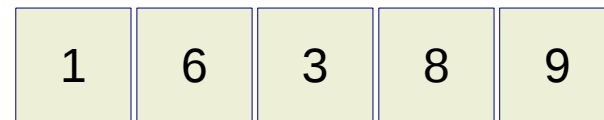
map(s -> s.substring(1))

"a9"->"a9".substring(1)



map(s -> Integer.parseInt(s))

"9"->Integer.parseInt("9")

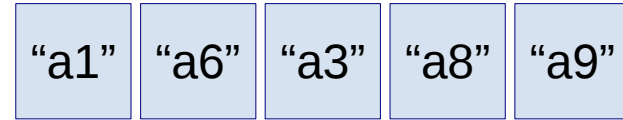


filter(i -> i > 6)

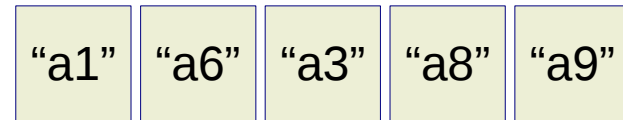
6->6>6

Results false. Nothing is added to result set

String sa[] = {"a1", "a6", "a3", "a8", "a9"};

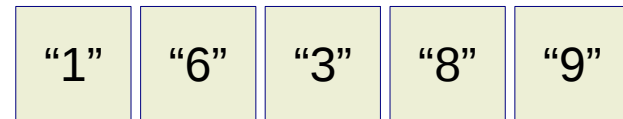


Arrays.stream(sa)



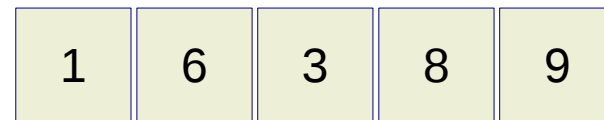
map(s -> s.substring(1))

"a9"->"a9".substring(1)



map(s -> Integer.parseInt(s))

"9"->Integer.parseInt("9")

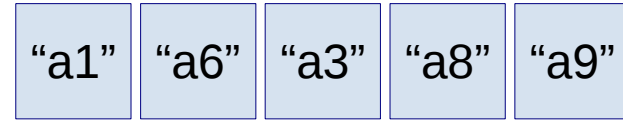


filter(i -> i > 6)

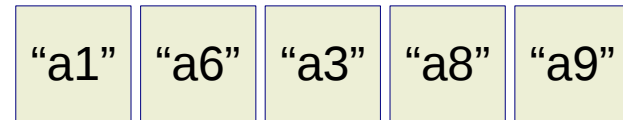
3->3>6

Results false. Nothing is added to result set

String sa[] = {"a1", "a6", "a3", "a8", "a9"};

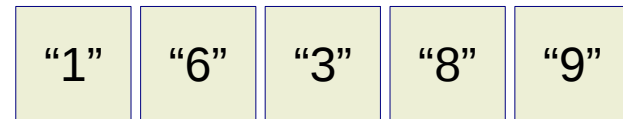


Arrays.stream(sa)



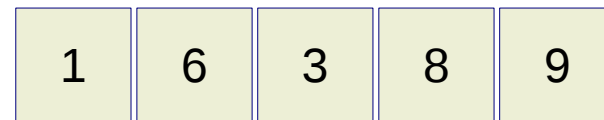
map(s -> s.substring(1))

"a9"->"a9".substring(1)



map(s -> Integer.parseInt(s))

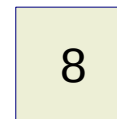
"9"->Integer.parseInt("9")



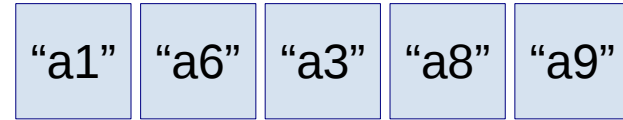
filter(i -> i > 6)

8->8>6

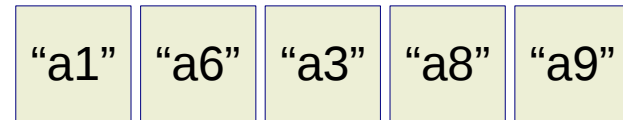
Results true. 8 is added to result set



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

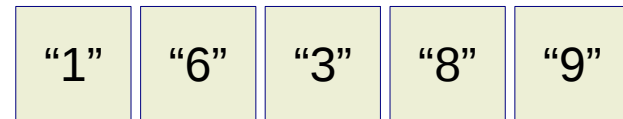


Arrays.stream(sa)



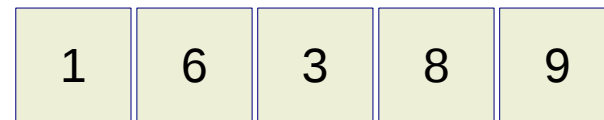
map(s -> s.substring(1))

"a9"->"a9".substring(1)



map(s -> Integer.parseInt(s))

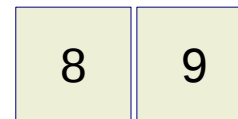
"9"->Integer.parseInt("9")



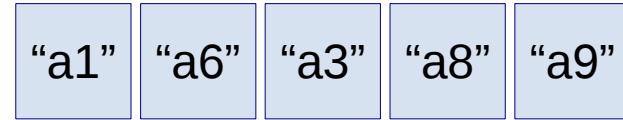
filter(i -> i > 6)

9->9>6

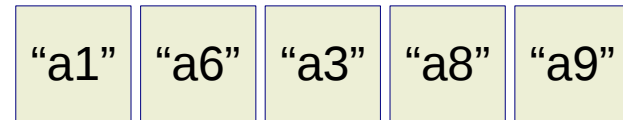
Results true. 9 is added to result set



String sa[] = {"a1", "a6", "a3", "a8", "a9"};

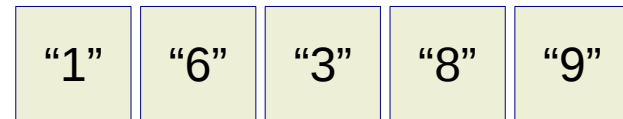


Arrays.stream(sa)



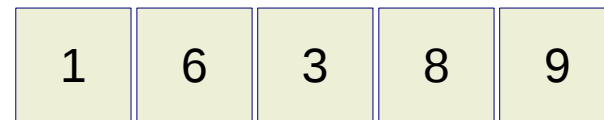
map(s -> s.substring(1))

"a9"->"a9".substring(1)



map(s -> Integer.parseInt(s))

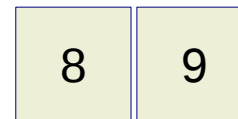
"9"->Integer.parseInt("9")



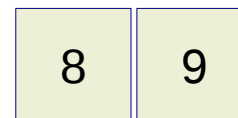
filter(i -> i > 6)

9->9>6

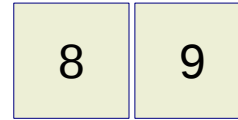
Results true. 9 is added to result set



reduce(0, Integer::sum);



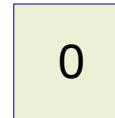
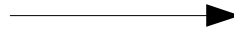
`reduce(0, Integer::sum);`



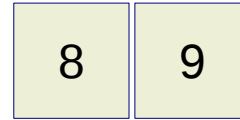
`Integer::sum` is a `BiFunction`

```
public static int sum(int a,int b)
```

Initialize result with 0



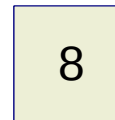
`reduce(0, Integer::sum);`



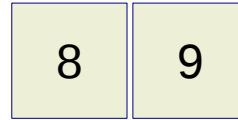
`Integer::sum` is a `BiFunction`

```
public static int sum(int a, int b)
```

`Integer.sum(0, 8)`



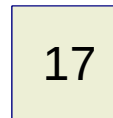
`reduce(0, Integer::sum);`



`Integer::sum` is a `BiFunction`

```
public static int sum(int a,int b)
```

`Integer.sum(8,9)`



```
reduce(0, Integer::sum);
```

8

9

`Integer::sum` is a `BiFunction`

```
public static int sum(int a,int b)
```

```
int summe
```



17