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

- 代表functional interface
- 可用于variables
- 可多次执行
- 可获取final variables(在le内部，可以使用在其外部定义的final变量)
- 可像data一样pass to methods（通常用於需要函數式接口的地方，比如 Comparator、Runnable或自定義的接口。）

```java
public class LambdaExample {
    public static void main(String[] args) {
        // 定義一個 final 變量
        final int number = 5;

        // 或者定義一個「有效 final 變量」
        int anotherNumber = 10;

        // 使用 lambda 表達式
        Runnable runnable = () -> {
            System.out.println("The number is: " + number);
            System.out.println("Another number is: " + anotherNumber);
        };

        // 試圖改變變量的值（如果取消註釋這行代碼，編譯器會報錯）
        // anotherNumber = 20;

        // 運行 lambda 表達式
        runnable.run();
    }
}
```

```java
public class LambdaAsParameterExample {
    public static void main(String[] args) {
        // 使用 lambda 表達式作為 Runnable 的實現
        Runnable task = () -> {
            System.out.println("Task is running");
        };

        // 將 lambda 表達式作為參數傳遞給方法
        process(task);
    }

    // 接受 Runnable 參數的方法
    public static void process(Runnable runnable) {
        System.out.println("Processing...");
        runnable.run();
    }
}
```

Before Java 8 Java instances where 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")
);
```

`e`代表一个ActionAvent对象，是表达式的参数，當按鈕被點擊時，這個對象會被傳遞給表達式。

这句表达式：用于为`JButton`注册一个`ActionListener`

OLD: Previously (inner) classes which implement a certain interface where used as method parameters

```
JButton testButton = new JButton("Test Button"); //same

testButton.addActionListener(new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent e){
          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());
```

Example Comparator

Example Comparable:

```
List<String> stringList = new ArrayList<String>();
stringList.add("John");
..

Collections.sort(stringList,
                 (String s1, String s2) -> s2.compareTo(s1));
```

Defining lamda expressions

Lambda syntax:

只要没有type，就不要加()，
除非是空的。
其它情况都要加()

```
input arguments -> 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 {}

Lamda type identification

ld表达式有特定类型的值，非随机代码片段

Lamda expressions have a type.

Type of expression is an instance of a functional interface.

其类型是一个函数式接口的实例

The interface is deduced from the context of the expression.

当你使用ld表达式时，
Java编译器会根据它所在的上下文
推断出它对应的函数式接口

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.

ld表达式必须具有与函数式接口的抽象方法相同的参数类型和返回类型。
也就是说，lambda表达式的签名（参数和返回类型）必须匹配它实现的函数式接口的唯一抽象方法。

Examples

Lambda

```
x -> x*2
```

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

Lambda

```
x -> {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`

```
// Supplier<T>: 不接受参数，返回一个类型T的结果

import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<String> supplier = () -> "Hello, World!";
        System.out.println(supplier.get());  // 输出: Hello, World!
    }
}

// Consumer<T>: 接受一个类型T的参数，不返回结果

public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> consumer = s -> System.out.println(s);
        consumer.accept("Hello, World!");  // 输出: Hello, World!
    }
}

// Predicate<T>: 接受一个类型T的参数，返回一个布尔值

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> predicate = n -> n > 5;
        System.out.println(predicate.test(10));  // 输出: true
        System.out.println(predicate.test(3));   // 输出: false
    }
}
```

Create your own interfaces

Create an interface with a **s**ingle **a**bstract **m**ethod. (SAM is another name for functional interfaces.)

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

创建了一个DumpPrinter接口的lambda实现,
并调用其doIt方法,
输出了"I am dumb"

Usage:

```
public class Main {
   public static void main(String[] args) {
       DumpPrinter dp = () -> System.out.println("I am dumb");
       dp.doIt();
    }
}
```

```
// iterface +
DumpPrinter dp = new DumpPrinter() {
    @Override
    public void doIt() {
    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 lamda 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

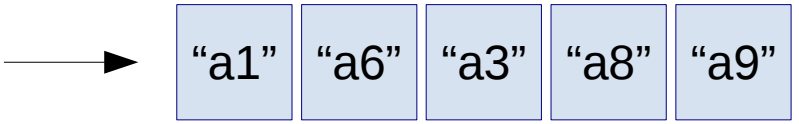
Function	Description
reduce()	Performs a reduction on the elements of this stream to a single value.
collect()	Groups the elements in this stream. You can use the Collectors from package `java.util.streams.Collectors`.
forEach()	Performs an action for each element of this list.

Most useful operations. For all operations see Streams javadoc

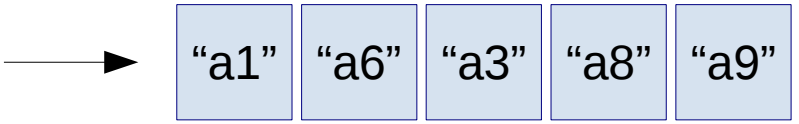
Streams revisited

```
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);
}
```

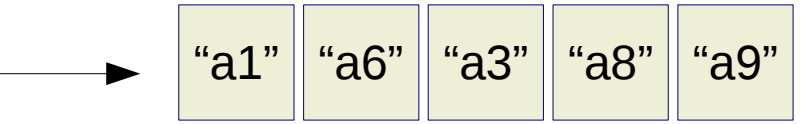
String sa[] = {"a1", "a6", "a3", "a8", "a9"};



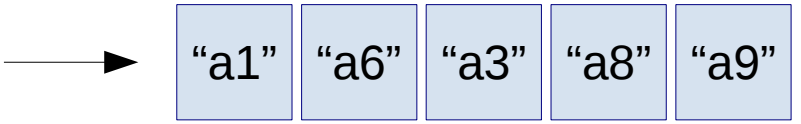
String sa[] = {"a1", "a6", "a3", "a8", "a9"};



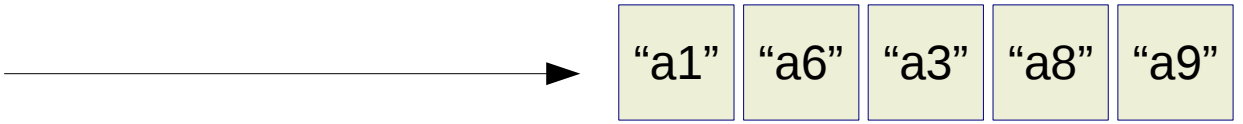
Arrays.stream(sa)



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



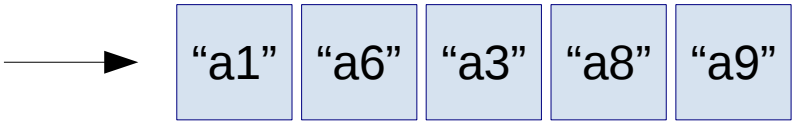
Arrays.stream(sa)



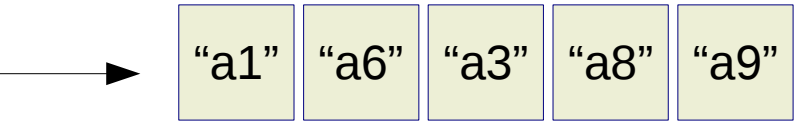
map(s -> s.substring(1))

“a1”->”a1”.substring(1)

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

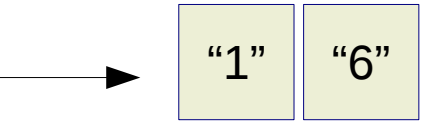


Arrays.stream(sa)

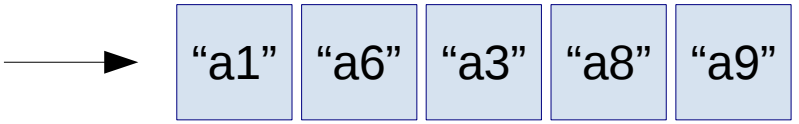


map(s -> s.substring(1))

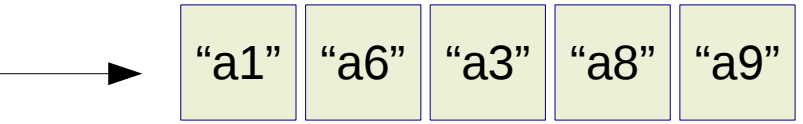
“a6”->”a6”.substring(1)



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

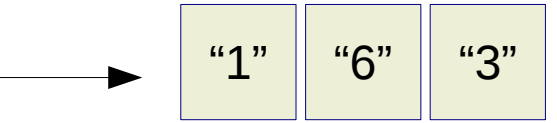


Arrays.stream(sa)

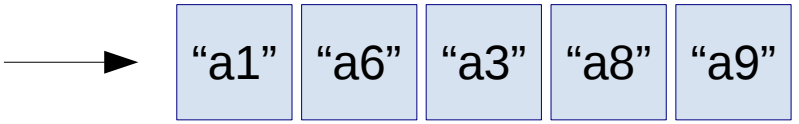


map(s -> s.substring(1))

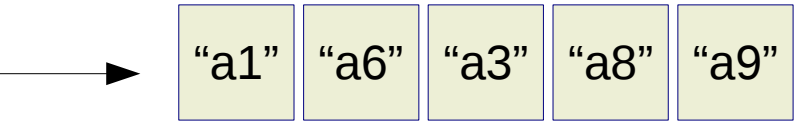
“a3”->”a3”.substring(1)



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

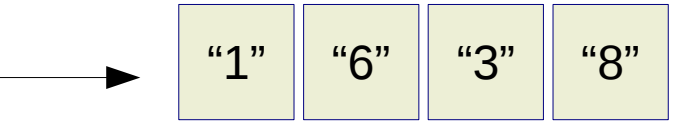


Arrays.stream(sa)

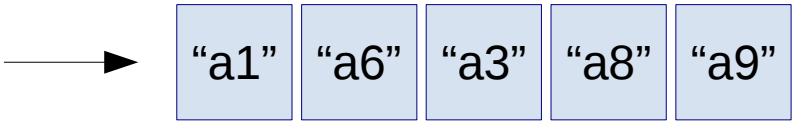


map(s -> s.substring(1))

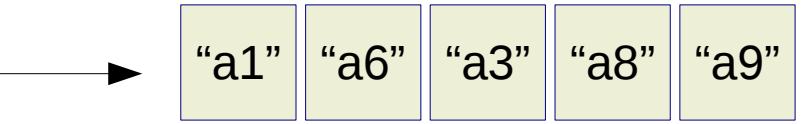
“a8”->”a8”.substring(1)



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

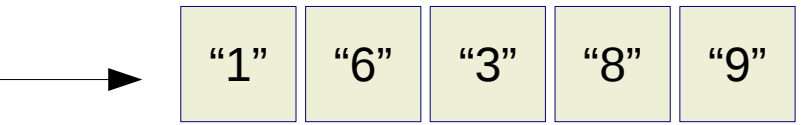


Arrays.stream(sa)



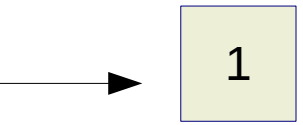
map(s -> s.substring(1))

“a9”->”a9”.substring(1)

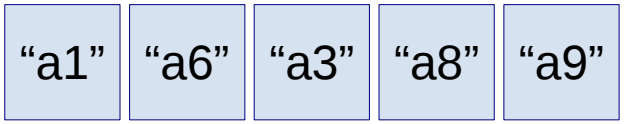


map(s -> Integer.parseInt(s))

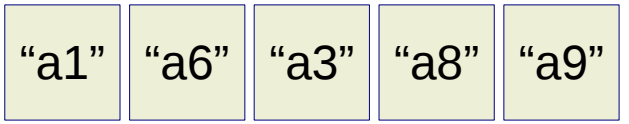
“1”->Integer.parseInt(“1”)



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

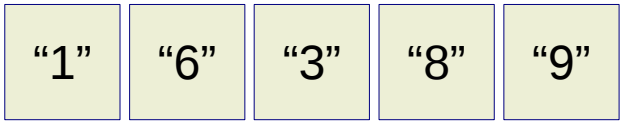


Arrays.stream(sa)



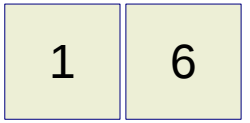
map(s -> s.substring(1))

“a9”->”a9”.substring(1)

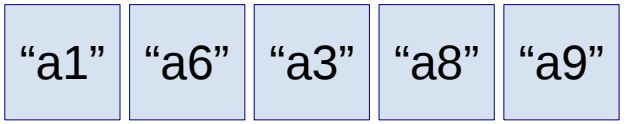


map(s -> Integer.parseInt(s))

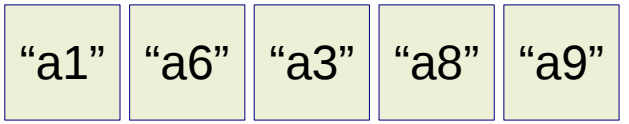
“6”->Integer.parseInt(“6”)



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

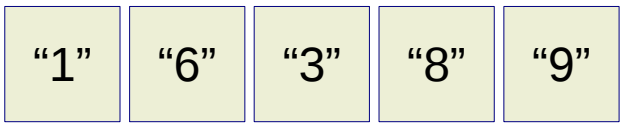


Arrays.stream(sa)



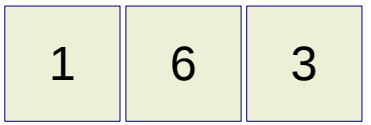
map(s -> s.substring(1))

“a9”->”a9”.substring(1)

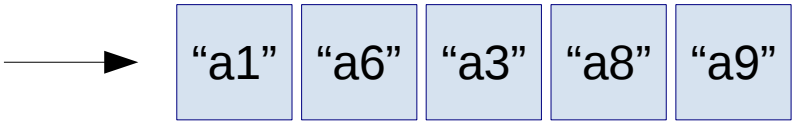


map(s -> Integer.parseInt(s))

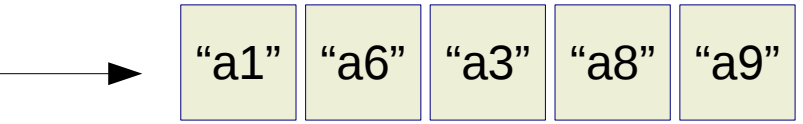
“3”->Integer.parseInt(“3”)



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

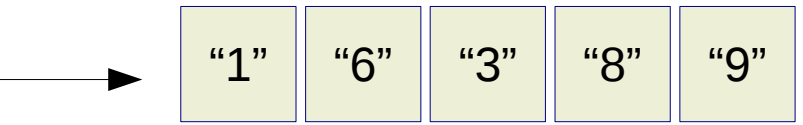


Arrays.stream(sa)



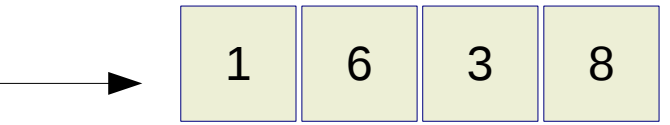
map(s -> s.substring(1))

“a9”->”a9”.substring(1)

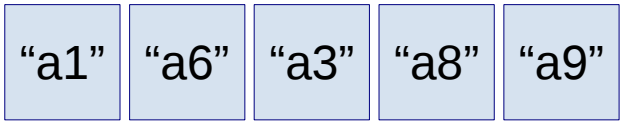


map(s -> Integer.parseInt(s))

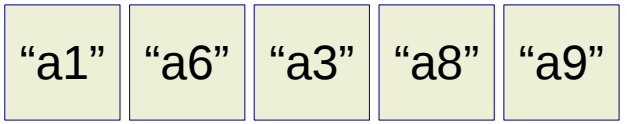
“8”->Integer.parseInt(“8”)



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

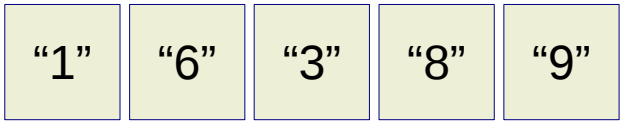


Arrays.stream(sa)



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

“a9”->”a9”.substring(1)



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

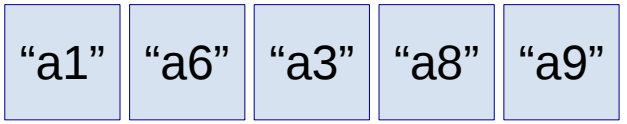
“9”->Integer.parseInt(“9”)

filter(i -> i > 6)

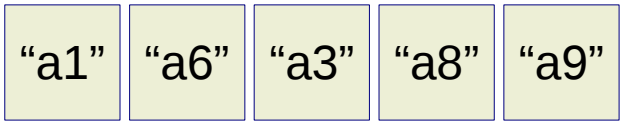
1->1>6

Results false. Nothing is added to result set

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

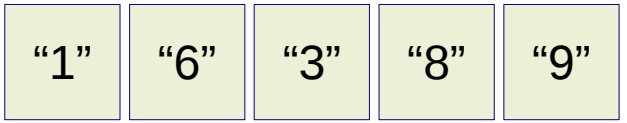


Arrays.stream(sa)



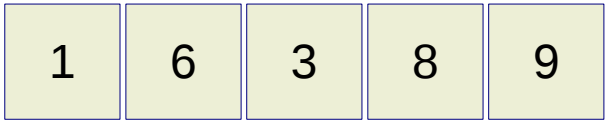
map(s -> s.substring(1))

“a9”->”a9”.substring(1)



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

“9”->Integer.parseInt(“9”)

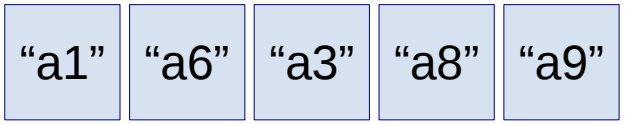


filter(i -> i > 6)

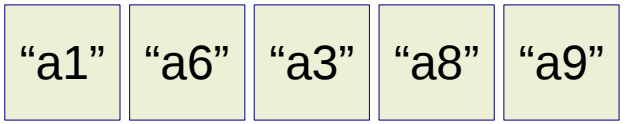
6->6>6

Results false. Nothing is added to result set

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

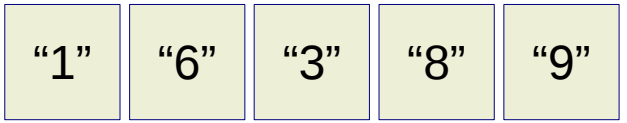


Arrays.stream(sa)



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

“a9”->”a9”.substring(1)



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

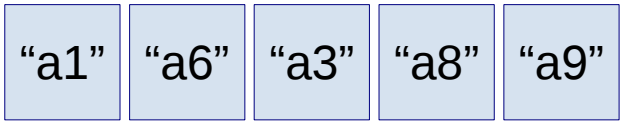
“9”->Integer.parseInt(“9”)

filter(i -> i > 6)

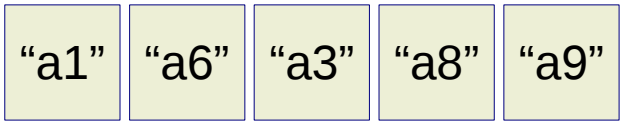
3->3>6

Results false. Nothing is added to result set

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

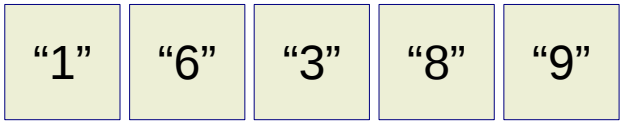


Arrays.stream(sa)



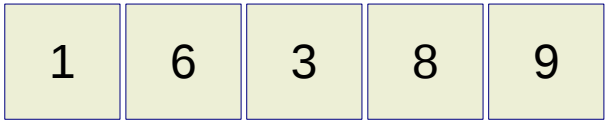
map(s -> s.substring(1))

“a9”->”a9”.substring(1)



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

“9”->Integer.parseInt(“9”)

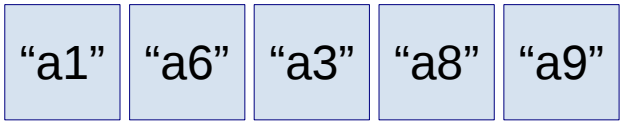


filter(i -> i > 6)

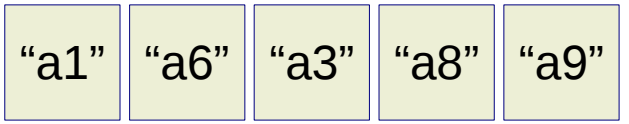
8->8>6

Results true. 8 is added to result set

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

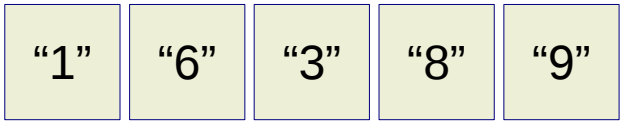


Arrays.stream(sa)



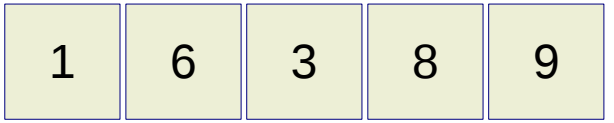
map(s -> s.substring(1))

“a9”->”a9”.substring(1)



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

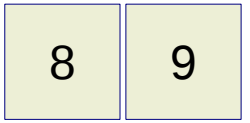
“9”->Integer.parseInt(“9”)



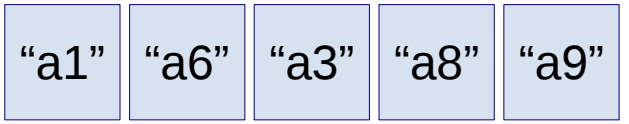
filter(i -> i > 6)

9->9>6

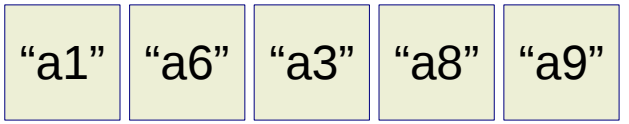
Results true. 9 is added to result set



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

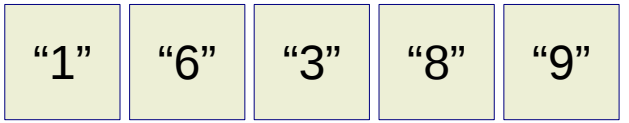


Arrays.stream(sa)



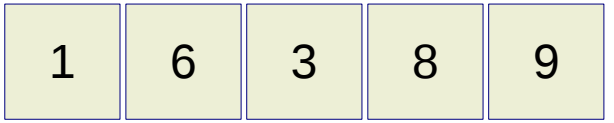
map(s -> s.substring(1))

“a9”->”a9”.substring(1)



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

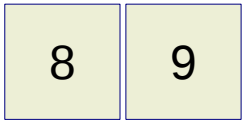
“9”->Integer.parseInt(“9”)



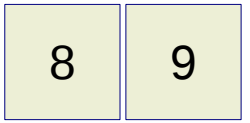
filter(i -> i > 6)

9->9>6

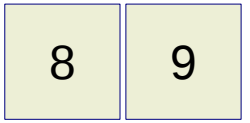
Results true. 9 is added to result set



reduce(0, Integer::sum);



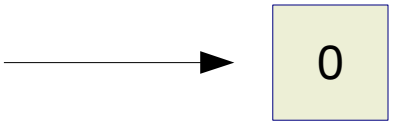
reduce(0, Integer::sum);



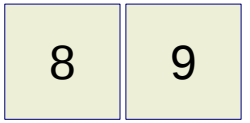
`Integer::sum` is a `BiFunction`

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

Initialize result with 0

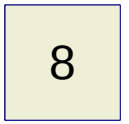


reduce(0, Integer::sum);

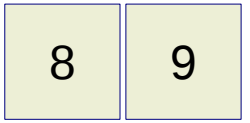


`Integer::sum` is a `BiFunction`

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

`Integer.sum(0,8)` → 

reduce(0, Integer::sum);

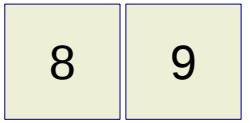


`Integer::sum` is a `BiFunction`

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

`Integer.sum(8,9)` → 17

reduce(0, Integer::sum);



`Integer::sum` is a `BiFunction`

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

`int summe`