Module java.base
Package java.util

Class Scanner

java.lang.Object
    java.util.Scanner

All Implemented Interfaces:

`Closeable`, `AutoCloseable`, `Iterator<String>`

```
public final class Scanner
extends Object
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

For example, this code allows a user to read a number from `System.in`:

```
     Scanner sc = new Scanner(System.in);
     int i = sc.nextInt();
```

As another example, this code allows `long` types to be assigned from entries in a file `myNumbers`:

```
      Scanner sc = new Scanner(new File("myNumbers"));
      while (sc.hasNextLong()) {
          long aLong = sc.nextLong();
      }
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

```
     String input = "1 fish 2 fish red fish blue fish";
     Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
     System.out.println(s.nextInt());
     System.out.println(s.nextInt());
     System.out.println(s.next());
     System.out.println(s.next());
     s.close();
```

prints the following output:

```
1
2
red
blue
```

The same output can be generated with this code, which uses a regular expression to parse all four tokens at once:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++)
    System.out.println(result.group(i));
s.close();
```

The default whitespace delimiter used by a scanner is as recognized by `Character.isWhitespace()`. The `reset()` method will reset the value of the scanner's delimiter to the default whitespace delimiter regardless of whether it was previously changed.

A scanning operation may block waiting for input.

The `next()` and `hasNext()` methods and their companion methods (such as `nextInt()` and `hasNextInt()`) first skip any input that matches the delimiter pattern, and then attempt to return the next token. Both `hasNext()` and `next()` methods may block waiting for further input. Whether a `hasNext()` method blocks has no connection to whether or not its associated `next()` method will block. The `tokens()` method may also block waiting for input.

The `findInLine()`, `findWithinHorizon()`, `skip()`, and `findAll()` methods operate independently of the delimiter pattern. These methods will attempt to match the specified pattern with no regard to delimiters in the input and thus can be used in special circumstances where delimiters are not relevant. These methods may block waiting for more input.

When a scanner throws an `InputMismatchException`, the scanner will not pass the token that caused the exception, so that it may be retrieved or skipped via some other method.

Depending upon the type of delimiting pattern, empty tokens may be returned. For example, the pattern `"\\s+"` will return no empty tokens since it matches multiple instances of the delimiter. The delimiting pattern `"\\s"` could return empty tokens since it only passes one space at a time.

A scanner can read text from any object which implements the `Readable` interface. If an invocation of the underlying readable's `read()` method throws an `IOException` then the scanner assumes that the end of the input has been reached. The most recent `IOException` thrown by the underlying readable can be retrieved via the `ioException()` method.

When a `Scanner` is closed, it will close its input source if the source implements the `Closeable` interface.

A `Scanner` is not safe for multithreaded use without external synchronization.

Unless otherwise mentioned, passing a `null` parameter into any method of a `Scanner` will cause a `NullPointerException` to be thrown.

A scanner will default to interpreting numbers as decimal unless a different radix has been set by using the `useRadix(int)` method. The `reset()` method will reset the value of the scanner's radix to `10` regardless of whether it was previously changed.

Localized numbers

An instance of this class is capable of scanning numbers in the standard formats as well as in the formats of the scanner's locale. A scanner's initial locale is the value returned by the `Locale.getDefault(Locale.Category.FORMAT)` method; it may be changed via the `useLocale()` method. The `reset()` method will reset the value of the scanner's locale to the initial locale regardless of whether it was previously changed.

The localized formats are defined in terms of the following parameters, which for a particular locale are taken from that locale's `DecimalFormat` object, `df`, and its and `DecimalFormatSymbols` object, `dfs`.

LocalGroupSeparator
    The character used to separate thousands groups, *i.e.,* `dfs.getGroupingSeparator()`
LocalDecimalSeparator
    The character used for the decimal point, *i.e.,* `dfs.getDecimalSeparator()`
LocalPositivePrefix
    The string that appears before a positive number (may be empty), *i.e.,* `df.getPositivePrefix()`
LocalPositiveSuffix
    The string that appears after a positive number (may be empty), *i.e.,* `df.getPositiveSuffix()`
LocalNegativePrefix
    The string that appears before a negative number (may be empty), *i.e.,* `df.getNegativePrefix()`
LocalNegativeSuffix
    The string that appears after a negative number (may be empty), *i.e.,* `df.getNegativeSuffix()`
LocalNaN
    The string that represents not-a-number for floating-point values, *i.e.,* `dfs.getNaN()`
LocalInfinity
    The string that represents infinity for floating-point values, *i.e.,* `dfs.getInfinity()`

Number syntax

The strings that can be parsed as numbers by an instance of this class are specified in terms of the following regular-expression grammar, where Rmax is the highest digit in the radix being used (for example, Rmax is 9 in base 10).

NonAsciiDigit:
    A non-ASCII character c for which `Character.isDigit(c)` returns true
Non0Digit:
    `[1-`*Rmax*`] |` *NonASCIIDigit*
Digit:
    `[0-`*Rmax*`] |` *NonASCIIDigit*

```
GroupedNumeral:
      ( Non0Digit Digit? Digit?
        ( LocalGroupSeparator Digit Digit Digit )+ )
Numeral:
      ( ( Digit+ ) | GroupedNumeral )
Integer:
      ( [-+]? ( Numeral ) )
      | LocalPositivePrefix Numeral LocalPositiveSuffix
      | LocalNegativePrefix Numeral LocalNegativeSuffix
DecimalNumeral:
      Numeral
      | Numeral LocalDecimalSeparator Digit*
      | LocalDecimalSeparator Digit+
Exponent:
      ( [eE] [+-]? Digit+ )
Decimal:
      ( [-+]? DecimalNumeral Exponent? )
      | LocalPositivePrefix DecimalNumeral LocalPositiveSuffix Exponent?
      | LocalNegativePrefix DecimalNumeral LocalNegativeSuffix Exponent?
HexFloat:
      [-+]? 0[xX][0-9a-fA-F]*\.[0-9a-fA-F]+ ([pP][-+]?[0-9]+)?
NonNumber:
      NaN | LocalNan | Infinity | LocalInfinity
SignedNonNumber:
      ( [-+]? NonNumber )
      | LocalPositivePrefix NonNumber LocalPositiveSuffix
      | LocalNegativePrefix NonNumber LocalNegativeSuffix
Float:
      Decimal | HexFloat | SignedNonNumber
```

Whitespace is not significant in the above regular expressions.

Since:

1.5

Constructor Summary

Constructors

Constructor	Description
`Scanner(File source)`	Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(File source, String charsetName)`	Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(File source, Charset charset)`	Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(InputStream source)`	Constructs a new `Scanner` that produces values scanned from the specified input stream.

`Scanner(InputStream source, String charsetName)`	Constructs a new `Scanner` that produces values scanned from the specified input stream.
`Scanner(InputStream source, Charset charset)`	Constructs a new `Scanner` that produces values scanned from the specified input stream.
`Scanner(Readable source)`	Constructs a new `Scanner` that produces values scanned from the specified source.
`Scanner(String source)`	Constructs a new `Scanner` that produces values scanned from the specified string.
`Scanner(ReadableByteChannel source)`	Constructs a new `Scanner` that produces values scanned from the specified channel.
`Scanner(ReadableByteChannel source, String charsetName)`	Constructs a new `Scanner` that produces values scanned from the specified channel.
`Scanner(ReadableByteChannel source, Charset charset)`	Constructs a new `Scanner` that produces values scanned from the specified channel.
`Scanner(Path source)`	Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(Path source, String charsetName)`	Constructs a new `Scanner` that produces values scanned from the specified file.
`Scanner(Path source, Charset charset)`	Constructs a new `Scanner` that produces values scanned from the specified file.

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
`void`	`close()`	Closes this scanner.
`Pattern`	`delimiter()`	Returns the `Pattern` this `Scanner` is currently using to match delimiters.
`Stream<MatchResult>`	`findAll(String patString)`	Returns a stream of match results that match the provided pattern string.
`Stream<MatchResult>`	`findAll(Pattern pattern)`	Returns a stream of match results from this scanner.

String	findInLine(String pattern)	Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
String	findInLine(Pattern pattern)	Attempts to find the next occurrence of the specified pattern ignoring delimiters.
String	findWithinHorizon (String pattern, int horizon)	Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
String	findWithinHorizon (Pattern pattern, int horizon)	Attempts to find the next occurrence of the specified pattern.
boolean	hasNext()	Returns true if this scanner has another token in its input.
boolean	hasNext(String pattern)	Returns true if the next token matches the pattern constructed from the specified string.
boolean	hasNext(Pattern pattern)	Returns true if the next complete token matches the specified pattern.
boolean	hasNextBigDecimal()	Returns true if the next token in this scanner's input can be interpreted as a `BigDecimal` using the `nextBigDecimal()` method.
boolean	hasNextBigInteger()	Returns true if the next token in this scanner's input can be interpreted as a `BigInteger` in the default radix using the `nextBigInteger()` method.
boolean	hasNextBigInteger (int radix)	Returns true if the next token in this scanner's input can be interpreted as a `BigInteger` in the specified radix using the `nextBigInteger()` method.
boolean	hasNextBoolean()	Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive

		pattern created from the string "true|false".
boolean	**hasNextByte**()	Returns true if the next token in this scanner's input can be interpreted as a byte value in the default radix using the nextByte() method.
boolean	**hasNextByte**(int radix)	Returns true if the next token in this scanner's input can be interpreted as a byte value in the specified radix using the nextByte() method.
boolean	**hasNextDouble**()	Returns true if the next token in this scanner's input can be interpreted as a double value using the nextDouble() method.
boolean	**hasNextFloat**()	Returns true if the next token in this scanner's input can be interpreted as a float value using the nextFloat() method.
boolean	**hasNextInt**()	Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the nextInt() method.
boolean	**hasNextInt**(int radix)	Returns true if the next token in this scanner's input can be interpreted as an int value in the specified radix using the nextInt() method.
boolean	**hasNextLine**()	Returns true if there is another line in the input of this scanner.
boolean	**hasNextLong**()	Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the nextLong() method.
boolean	**hasNextLong**(int radix)	Returns true if the next token in this scanner's input can be interpreted as a long value in the specified radix using the nextLong() method.

boolean	**hasNextShort**()	Returns true if the next token in this scanner's input can be interpreted as a short value in the default radix using the `nextShort()` method.
boolean	**hasNextShort**(int radix)	Returns true if the next token in this scanner's input can be interpreted as a short value in the specified radix using the `nextShort()` method.
IOException	**ioException**()	Returns the `IOException` last thrown by this `Scanner`'s underlying `Readable`.
Locale	**locale**()	Returns this scanner's locale.
MatchResult	**match**()	Returns the match result of the last scanning operation performed by this scanner.
String	**next**()	Finds and returns the next complete token from this scanner.
String	**next**(**String** pattern)	Returns the next token if it matches the pattern constructed from the specified string.
String	**next**(**Pattern** pattern)	Returns the next token if it matches the specified pattern.
BigDecimal	**nextBigDecimal**()	Scans the next token of the input as a `BigDecimal`.
BigInteger	**nextBigInteger**()	Scans the next token of the input as a `BigInteger`.
BigInteger	**nextBigInteger**(int radix)	Scans the next token of the input as a `BigInteger`.
boolean	**nextBoolean**()	Scans the next token of the input into a boolean value and returns that value.
byte	**nextByte**()	Scans the next token of the input as a `byte`.
byte	**nextByte**(int radix)	Scans the next token of the input as a `byte`.
double	**nextDouble**()	Scans the next token of the input as a `double`.

`float`	`nextFloat()`	Scans the next token of the input as a `float`.
`int`	`nextInt()`	Scans the next token of the input as an `int`.
`int`	`nextInt(int radix)`	Scans the next token of the input as an `int`.
`String`	`nextLine()`	Advances this scanner past the current line and returns the input that was skipped.
`long`	`nextLong()`	Scans the next token of the input as a `long`.
`long`	`nextLong(int radix)`	Scans the next token of the input as a `long`.
`short`	`nextShort()`	Scans the next token of the input as a `short`.
`short`	`nextShort(int radix)`	Scans the next token of the input as a `short`.
`int`	`radix()`	Returns this scanner's default radix.
`void`	`remove()`	The remove operation is not supported by this implementation of `Iterator`.
`Scanner`	`reset()`	Resets this scanner.
`Scanner`	`skip(String pattern)`	Skips input that matches a pattern constructed from the specified string.
`Scanner`	`skip(Pattern pattern)`	Skips input that matches the specified pattern, ignoring delimiters.
`Stream<String>`	`tokens()`	Returns a stream of delimiter-separated tokens from this scanner.
`String`	`toString()`	Returns the string representation of this `Scanner`.
`Scanner`	`useDelimiter(String pattern)`	Sets this scanner's delimiting pattern to a pattern constructed from the specified `String`.

`Scanner`	`useDelimiter(Pattern pattern)`	Sets this scanner's delimiting pattern to the specified pattern.
`Scanner`	`useLocale(Locale locale)`	Sets this scanner's locale to the specified locale.
`Scanner`	`useRadix(int radix)`	Sets this scanner's default radix to the specified radix.

Methods declared in class java.lang.Object

`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

Methods declared in interface java.util.Iterator

`forEachRemaining`

Constructor Details

Scanner

```
public Scanner(Readable source)
```

Constructs a new `Scanner` that produces values scanned from the specified source.

Parameters:

`source` - A character source implementing the `Readable` interface

Scanner

```
public Scanner(InputStream source)
```

Constructs a new `Scanner` that produces values scanned from the specified input stream. Bytes from the stream are converted into characters using the underlying platform's default charset.

Parameters:

`source` - An input stream to be scanned

Scanner

```
public Scanner(InputStream source,
               String charsetName)
```

Constructs a new `Scanner` that produces values scanned from the specified input stream. Bytes from the stream are converted into characters using the specified charset.

Parameters:

`source` - An input stream to be scanned

`charsetName` - The encoding type used to convert bytes from the stream into characters to be scanned

Throws:

`IllegalArgumentException` - if the specified character set does not exist

Scanner

```
public Scanner(InputStream source,
               Charset charset)
```

Constructs a new `Scanner` that produces values scanned from the specified input stream. Bytes from the stream are converted into characters using the specified charset.

Parameters:

`source` - an input stream to be scanned

`charset` - the charset used to convert bytes from the file into characters to be scanned

Since:

10

Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

`source` - A file to be scanned

Throws:

`FileNotFoundException` - if source is not found

Scanner

```
public Scanner(File source,
               String charsetName)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the specified charset.

Parameters:

`source` - A file to be scanned

`charsetName` - The encoding type used to convert bytes from the file into characters to be scanned

Throws:

`FileNotFoundException` - if source is not found

`IllegalArgumentException` - if the specified encoding is not found

Scanner

```
public Scanner(File source,
               Charset charset)
        throws IOException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the specified charset.

Parameters:

`source` - A file to be scanned

`charset` - The charset used to convert bytes from the file into characters to be scanned

Throws:

`IOException` - if an I/O error occurs opening the source

Since:

10

Scanner

```
public Scanner(Path source)
        throws IOException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

`source` - the path to the file to be scanned

Throws:

`IOException` - if an I/O error occurs opening source

Since:

1.7

Scanner

```
public Scanner(Path source,
               String charsetName)
        throws IOException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the specified charset.

Parameters:

`source` - the path to the file to be scanned

`charsetName` - The encoding type used to convert bytes from the file into characters to be scanned

Throws:

`IOException` - if an I/O error occurs opening source

`IllegalArgumentException` - if the specified encoding is not found

Since:

1.7

Scanner

```
public Scanner(Path source,
               Charset charset)
        throws IOException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the specified charset.

Parameters:

`source` - the path to the file to be scanned

`charset` - the charset used to convert bytes from the file into characters to be scanned

Throws:

`IOException` - if an I/O error occurs opening the source

Since:

10

Scanner

```
public Scanner(String source)
```

Constructs a new `Scanner` that produces values scanned from the specified string.

Parameters:

`source` - A string to scan

Scanner

```
public Scanner(ReadableByteChannel source)
```

Constructs a new `Scanner` that produces values scanned from the specified channel. Bytes from the source are converted into characters using the underlying platform's default charset.

Parameters:

`source` - A channel to scan

Scanner

```
public Scanner(ReadableByteChannel source,
               String charsetName)
```

Constructs a new `Scanner` that produces values scanned from the specified channel. Bytes from the source are converted into characters using the specified charset.

Parameters:

`source` - A channel to scan

`charsetName` - The encoding type used to convert bytes from the channel into characters to be scanned

Throws:

`IllegalArgumentException` - if the specified character set does not exist

Scanner

```
public Scanner(ReadableByteChannel source,
               Charset charset)
```

Constructs a new `Scanner` that produces values scanned from the specified channel. Bytes from the source are converted into characters using the specified charset.

Parameters:

`source` - a channel to scan

`charset` - the encoding type used to convert bytes from the channel into characters to be scanned

Since:

10

Method Details

close

```
public void close()
```

Closes this scanner.

If this scanner has not yet been closed then if its underlying readable also implements the `Closeable` interface then the readable's `close` method will be invoked. If this scanner is already closed then invoking this method will have no effect.

Attempting to perform search operations after a scanner has been closed will result in an `IllegalStateException`.

Specified by:

`close` in interface `AutoCloseable`

Specified by:

`close` in interface `Closeable`

ioException

```
public IOException ioException()
```

Returns the `IOException` last thrown by this `Scanner`'s underlying `Readable`. This method returns `null` if no such exception exists.

Returns:

the last exception thrown by this scanner's readable

delimiter

```
public Pattern delimiter()
```

Returns the `Pattern` this `Scanner` is currently using to match delimiters.

Returns:

this scanner's delimiting pattern.

useDelimiter

```
public Scanner useDelimiter(Pattern pattern)
```

Sets this scanner's delimiting pattern to the specified pattern.

Parameters:

`pattern` - A delimiting pattern

Returns:

this scanner

useDelimiter

```
public Scanner useDelimiter(String pattern)
```

Sets this scanner's delimiting pattern to a pattern constructed from the specified `String`.

An invocation of this method of the form `useDelimiter(pattern)` behaves in exactly the same way as the invocation `useDelimiter(Pattern.compile(pattern))`.

Invoking the `reset()` method will set the scanner's delimiter to the default.

Parameters:

`pattern` - A string specifying a delimiting pattern

Returns:

this scanner

locale

```
public Locale locale()
```

Returns this scanner's locale.

A scanner's locale affects many elements of its default primitive matching regular expressions; see localized numbers above.

Returns:

this scanner's locale

useLocale

```
public Scanner useLocale(Locale locale)
```

Sets this scanner's locale to the specified locale.

A scanner's locale affects many elements of its default primitive matching regular expressions; see localized numbers above.

Invoking the `reset()` method will set the scanner's locale to the initial locale.

Parameters:

`locale` - A string specifying the locale to use

Returns:

this scanner

radix

```
public int radix()
```

Returns this scanner's default radix.

A scanner's radix affects elements of its default number matching regular expressions; see localized numbers above.

Returns:

the default radix of this scanner

useRadix

```
public Scanner useRadix(int radix)
```

Sets this scanner's default radix to the specified radix.

A scanner's radix affects elements of its default number matching regular expressions; see localized numbers above.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Invoking the `reset()` method will set the scanner's radix to `10`.

Parameters:

`radix` - The radix to use when scanning numbers

Returns:

this scanner

Throws:

`IllegalArgumentException` - if radix is out of range

match

```
public MatchResult match()
```

Returns the match result of the last scanning operation performed by this scanner. This method throws `IllegalStateException` if no match has been performed, or if the last match was not successful.

The various `next` methods of `Scanner` make a match result available if they complete without throwing an exception. For instance, after an invocation of the `nextInt()` method that returned an int, this method returns a `MatchResult` for the search of the *Integer* regular expression defined above. Similarly the `findInLine()`, `findWithinHorizon()`, and `skip()` methods will make a match available if they succeed.

Returns:

a match result for the last match operation

Throws:

`IllegalStateException` - If no match result is available

toString

```
public String toString()
```

Returns the string representation of this `Scanner`. The string representation of a `Scanner` contains information that may be useful for debugging. The exact format is unspecified.

Overrides:

`toString` in class `Object`

Returns:

The string representation of this scanner

hasNext

`public boolean hasNext()`

Returns true if this scanner has another token in its input. This method may block while waiting for input to scan. The scanner does not advance past any input.

Specified by:

`hasNext` in interface `Iterator<String>`

Returns:

true if and only if this scanner has another token

Throws:

`IllegalStateException` - if this scanner is closed

See Also:

`Iterator`

next

`public String next()`

Finds and returns the next complete token from this scanner. A complete token is preceded and followed by input that matches the delimiter pattern. This method may block while waiting for input to scan, even if a previous invocation of `hasNext()` returned `true`.

Specified by:

`next` in interface `Iterator<String>`

Returns:

the next token

Throws:

`NoSuchElementException` - if no more tokens are available

`IllegalStateException` - if this scanner is closed

See Also:

`Iterator`

remove

`public void remove()`

The remove operation is not supported by this implementation of `Iterator`.

Specified by:

`remove` in interface `Iterator<String>`

Throws:

`UnsupportedOperationException` - if this method is invoked.

See Also:

`Iterator`

hasNext

`public boolean hasNext(String pattern)`

Returns true if the next token matches the pattern constructed from the specified string. The scanner does not advance past any input.

An invocation of this method of the form `hasNext(pattern)` behaves in exactly the same way as the invocation `hasNext(Pattern.compile(pattern))`.

Parameters:

`pattern` - a string specifying the pattern to scan

Returns:

true if and only if this scanner has another token matching the specified pattern

Throws:

`IllegalStateException` - if this scanner is closed

next

`public String next(String pattern)`

Returns the next token if it matches the pattern constructed from the specified string. If the match is successful, the scanner advances past the input that matched the pattern.

An invocation of this method of the form `next(pattern)` behaves in exactly the same way as the invocation `next(Pattern.compile(pattern))`.

Parameters:

`pattern` - a string specifying the pattern to scan

Returns:

the next token

Throws:

`NoSuchElementException` - if no such tokens are available

`IllegalStateException` - if this scanner is closed

hasNext

`public boolean hasNext(Pattern pattern)`

Returns true if the next complete token matches the specified pattern. A complete token is prefixed and postfixed by input that matches the delimiter pattern. This method may block while waiting for input. The scanner does not advance past any input.

Parameters:

`pattern` - the pattern to scan for

Returns:

true if and only if this scanner has another token matching the specified pattern

Throws:

`IllegalStateException` - if this scanner is closed

next

```
public String next(Pattern pattern)
```

Returns the next token if it matches the specified pattern. This method may block while waiting for input to scan, even if a previous invocation of `hasNext(Pattern)` returned `true`. If the match is successful, the scanner advances past the input that matched the pattern.

Parameters:

`pattern` - the pattern to scan for

Returns:

the next token

Throws:

`NoSuchElementException` - if no more tokens are available

`IllegalStateException` - if this scanner is closed

hasNextLine

```
public boolean hasNextLine()
```

Returns true if there is another line in the input of this scanner. This method may block while waiting for input. The scanner does not advance past any input.

Returns:

true if and only if this scanner has another line of input

Throws:

`IllegalStateException` - if this scanner is closed

nextLine

```
public String nextLine()
```

Advances this scanner past the current line and returns the input that was skipped. This method returns the rest of the current line, excluding any line separator at the end. The position is set to the beginning of the next line.

Since this method continues to search through the input looking for a line separator, it may buffer all of the input searching for the line to skip if no line separators are present.

Returns:

the line that was skipped

Throws:

`NoSuchElementException` - if no line was found

`IllegalStateException` - if this scanner is closed

findInLine

```
public String findInLine(String pattern)
```

Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.

An invocation of this method of the form `findInLine(pattern)` behaves in exactly the same way as the invocation `findInLine(Pattern.compile(pattern))`.

Parameters:

`pattern` - a string specifying the pattern to search for

Returns:

the text that matched the specified pattern

Throws:

`IllegalStateException` - if this scanner is closed

findInLine

```
public String findInLine(Pattern pattern)
```

Attempts to find the next occurrence of the specified pattern ignoring delimiters. If the pattern is found before the next line separator, the scanner advances past the input that matched and returns the string that matched the pattern. If no such pattern is detected in the input up to the next line separator, then `null` is returned and the scanner's position is unchanged. This method may block waiting for input that matches the pattern.

Since this method continues to search through the input looking for the specified pattern, it may buffer all of the input searching for the desired token if no line separators are present.

Parameters:

`pattern` - the pattern to scan for

Returns:

the text that matched the specified pattern

Throws:

`IllegalStateException` - if this scanner is closed

findWithinHorizon

```
public String findWithinHorizon(String pattern,
                                int horizon)
```

Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.

An invocation of this method of the form `findWithinHorizon(pattern)` behaves in exactly the same way as the invocation `findWithinHorizon(Pattern.compile(pattern), horizon)`.

Parameters:

`pattern` - a string specifying the pattern to search for

`horizon` - the search horizon

Returns:

the text that matched the specified pattern

Throws:

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if horizon is negative

findWithinHorizon

```
public String findWithinHorizon(Pattern pattern,
                                int horizon)
```

Attempts to find the next occurrence of the specified pattern.

This method searches through the input up to the specified search horizon, ignoring delimiters. If the pattern is found the scanner advances past the input that matched and returns the string that matched the pattern. If no such pattern is detected then the null is returned and the scanner's position remains unchanged. This method may block waiting for input that matches the pattern.

A scanner will never search more than `horizon` code points beyond its current position. Note that a match may be clipped by the horizon; that is, an arbitrary match result may have been different if the horizon had been larger. The scanner treats the horizon as a transparent, non-anchoring bound (see `Matcher.useTransparentBounds(boolean)` and `Matcher.useAnchoringBounds(boolean)`).

If horizon is `0`, then the horizon is ignored and this method continues to search through the input looking for the specified pattern without bound. In this case it may buffer all of the input searching for the pattern.

If horizon is negative, then an IllegalArgumentException is thrown.

Parameters:

`pattern` - the pattern to scan for

`horizon` - the search horizon

Returns:

the text that matched the specified pattern

Throws:

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if horizon is negative

skip

```
public Scanner skip(Pattern pattern)
```

Skips input that matches the specified pattern, ignoring delimiters. This method will skip input if an anchored match of the specified pattern succeeds.

If a match to the specified pattern is not found at the current position, then no input is skipped and a `NoSuchElementException` is thrown.

Since this method seeks to match the specified pattern starting at the scanner's current position, patterns that can match a lot of input (".*", for example) may cause the scanner to buffer a large amount of input.

Note that it is possible to skip something without risking a `NoSuchElementException` by using a pattern that can match nothing, e.g., `sc.skip("[ \t]*")`.

Parameters:

`pattern` - a string specifying the pattern to skip over

Returns:

this scanner

Throws:

`NoSuchElementException` - if the specified pattern is not found

`IllegalStateException` - if this scanner is closed

skip

```
public Scanner skip(String pattern)
```

Skips input that matches a pattern constructed from the specified string.

An invocation of this method of the form `skip(pattern)` behaves in exactly the same way as the invocation `skip(Pattern.compile(pattern))`.

Parameters:

`pattern` - a string specifying the pattern to skip over

Returns:

this scanner

Throws:

`IllegalStateException` - if this scanner is closed

hasNextBoolean

```
public boolean hasNextBoolean()
```

Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false". The scanner does not advance past the input that matched.

Returns:

true if and only if this scanner's next token is a valid boolean value

Throws:

`IllegalStateException` - if this scanner is closed

nextBoolean

```
public boolean nextBoolean()
```

Scans the next token of the input into a boolean value and returns that value. This method will throw `InputMismatchException` if the next token cannot be translated into a valid boolean value. If the match is successful, the scanner advances past the input that matched.

Returns:

the boolean scanned from the input

Throws:

`InputMismatchException` - if the next token is not a valid boolean

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

hasNextByte

```
public boolean hasNextByte()
```

Returns true if the next token in this scanner's input can be interpreted as a byte value in the default radix using the `nextByte()` method. The scanner does not advance past any input.

Returns:

true if and only if this scanner's next token is a valid byte value

Throws:

`IllegalStateException` - if this scanner is closed

hasNextByte

```
public boolean hasNextByte(int radix)
```

Returns true if the next token in this scanner's input can be interpreted as a byte value in the specified radix using the `nextByte()` method. The scanner does not advance past any input.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as a byte value

Returns:

true if and only if this scanner's next token is a valid byte value

Throws:

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

nextByte

```
public byte nextByte()
```

Scans the next token of the input as a `byte`.

An invocation of this method of the form `nextByte()` behaves in exactly the same way as the invocation `nextByte(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `byte` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

nextByte

```
public byte nextByte(int radix)
```

Scans the next token of the input as a `byte`. This method will throw `InputMismatchException` if the next token cannot be translated into a valid byte value as described below. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Integer* regular expression defined above then the token is converted into a `byte` value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via `Character.digit`, prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to `Byte.parseByte` with the specified radix.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as a byte value

Returns:

the `byte` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

hasNextShort

`public boolean hasNextShort()`

Returns true if the next token in this scanner's input can be interpreted as a short value in the default radix using the `nextShort()` method. The scanner does not advance past any input.

Returns:

true if and only if this scanner's next token is a valid short value in the default radix

Throws:

`IllegalStateException` - if this scanner is closed

hasNextShort

`public boolean hasNextShort(int radix)`

Returns true if the next token in this scanner's input can be interpreted as a short value in the specified radix using the `nextShort()` method. The scanner does not advance past any input.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as a short value

Returns:

true if and only if this scanner's next token is a valid short value in the specified radix

Throws:

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

nextShort

`public short nextShort()`

Scans the next token of the input as a `short`.

An invocation of this method of the form `nextShort()` behaves in exactly the same way as the invocation `nextShort(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `short` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

nextShort

```
public short nextShort(int radix)
```

Scans the next token of the input as a `short`. This method will throw `InputMismatchException` if the next token cannot be translated into a valid short value as described below. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Integer* regular expression defined above then the token is converted into a `short` value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via `Character.digit`, prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to `Short.parseShort` with the specified radix.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as a short value

Returns:

the `short` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

hasNextInt

```
public boolean hasNextInt()
```

Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the `nextInt()` method. The scanner does not advance past any input.

Returns:

true if and only if this scanner's next token is a valid int value

Throws:

`IllegalStateException` - if this scanner is closed

hasNextInt

```
public boolean hasNextInt(int radix)
```

Returns true if the next token in this scanner's input can be interpreted as an int value in the specified radix using the `nextInt()` method. The scanner does not advance past any input.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as an int value

Returns:

true if and only if this scanner's next token is a valid int value

Throws:

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

nextInt

```
public int nextInt()
```

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `int` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

nextInt

```
public int nextInt(int radix)
```

Scans the next token of the input as an `int`. This method will throw `InputMismatchException` if the next token cannot be translated into a valid int value as described below. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Integer* regular expression defined above then the token is converted into an `int` value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via `Character.digit`, prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to `Integer.parseInt` with the specified radix.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as an int value

Returns:

the `int` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

hasNextLong

```
public boolean hasNextLong()
```

Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the `nextLong()` method. The scanner does not advance past any input.

Returns:

true if and only if this scanner's next token is a valid long value

Throws:

`IllegalStateException` - if this scanner is closed

hasNextLong

```
public boolean hasNextLong(int radix)
```

Returns true if the next token in this scanner's input can be interpreted as a long value in the specified radix using the `nextLong()` method. The scanner does not advance past any input.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as a long value

Returns:

true if and only if this scanner's next token is a valid long value

Throws:

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

nextLong

```
public long nextLong()
```

Scans the next token of the input as a `long`.

An invocation of this method of the form `nextLong()` behaves in exactly the same way as the invocation `nextLong(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `long` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

nextLong

```
public long nextLong(int radix)
```

Scans the next token of the input as a `long`. This method will throw `InputMismatchException` if the next token cannot be translated into a valid long value as described below. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Integer* regular expression defined above then the token is converted into a `long` value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via `Character.digit`, prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to `Long.parseLong` with the specified radix.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as an int value

Returns:

the `long` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

hasNextFloat

```
public boolean hasNextFloat()
```

Returns true if the next token in this scanner's input can be interpreted as a float value using the `nextFloat()` method. The scanner does not advance past any input.

Returns:

true if and only if this scanner's next token is a valid float value

Throws:

`IllegalStateException` - if this scanner is closed

nextFloat

```
public float nextFloat()
```

Scans the next token of the input as a `float`. This method will throw `InputMismatchException` if the next token cannot be translated into a valid float value as described below. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Float* regular expression defined above then the token is converted into a `float` value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via `Character.digit`, prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to `Float.parseFloat`. If the token matches the localized NaN or infinity strings, then either "Nan" or "Infinity" is passed to `Float.parseFloat` as appropriate.

Returns:

the `float` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Float* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

hasNextDouble

```
public boolean hasNextDouble()
```

Returns true if the next token in this scanner's input can be interpreted as a double value using the `nextDouble()` method. The scanner does not advance past any input.

Returns:

true if and only if this scanner's next token is a valid double value

Throws:

`IllegalStateException` - if this scanner is closed

nextDouble

```
public double nextDouble()
```

Scans the next token of the input as a `double`. This method will throw `InputMismatchException` if the next token cannot be translated into a valid double value. If the translation is successful, the scanner advances past the input that matched.

If the next token matches the *Float* regular expression defined above then the token is converted into a `double` value as if by removing all locale specific prefixes, group separators, and locale specific suffixes, then mapping non-ASCII digits into ASCII digits via `Character.digit`, prepending a negative sign (-) if the locale specific negative prefixes and suffixes were present, and passing the resulting string to `Double.parseDouble`. If the token matches the localized NaN or infinity strings, then either "Nan" or "Infinity" is passed to `Double.parseDouble` as appropriate.

Returns:

the `double` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Float* regular expression, or is out of range

`NoSuchElementException` - if the input is exhausted

`IllegalStateException` - if this scanner is closed

hasNextBigInteger

```
public boolean hasNextBigInteger()
```

Returns true if the next token in this scanner's input can be interpreted as a `BigInteger` in the default radix using the `nextBigInteger()` method. The scanner does not advance past any input.

Returns:

true if and only if this scanner's next token is a valid `BigInteger`

Throws:

`IllegalStateException` - if this scanner is closed

hasNextBigInteger

`public boolean hasNextBigInteger(int radix)`

Returns true if the next token in this scanner's input can be interpreted as a `BigInteger` in the specified radix using the `nextBigInteger()` method. The scanner does not advance past any input.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token as an integer

Returns:

true if and only if this scanner's next token is a valid `BigInteger`

Throws:

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

nextBigInteger

`public BigInteger nextBigInteger()`

Scans the next token of the input as a `BigInteger`.

An invocation of this method of the form `nextBigInteger()` behaves in exactly the same way as the invocation `nextBigInteger(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `BigInteger` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if the input is exhausted

`IllegalStateException` - if this scanner is closed

nextBigInteger

`public BigInteger nextBigInteger(int radix)`

Scans the next token of the input as a `BigInteger`.

If the next token matches the *Integer* regular expression defined above then the token is converted into a `BigInteger` value as if by removing all group separators, mapping non-ASCII digits into ASCII digits via the `Character.digit`, and passing the resulting string to the `BigInteger(String, int)` constructor with the specified radix.

If the radix is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, then an `IllegalArgumentException` is thrown.

Parameters:

`radix` - the radix used to interpret the token

Returns:

the `BigInteger` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if the input is exhausted

`IllegalStateException` - if this scanner is closed

`IllegalArgumentException` - if the radix is out of range

hasNextBigDecimal

```
public boolean hasNextBigDecimal()
```

Returns true if the next token in this scanner's input can be interpreted as a `BigDecimal` using the `nextBigDecimal()` method. The scanner does not advance past any input.

Returns:

true if and only if this scanner's next token is a valid `BigDecimal`

Throws:

`IllegalStateException` - if this scanner is closed

nextBigDecimal

```
public BigDecimal nextBigDecimal()
```

Scans the next token of the input as a `BigDecimal`.

If the next token matches the *Decimal* regular expression defined above then the token is converted into a `BigDecimal` value as if by removing all group separators, mapping non-ASCII digits into ASCII digits via the `Character.digit`, and passing the resulting string to the `BigDecimal(String)` constructor.

Returns:

the `BigDecimal` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Decimal* regular expression, or is out of range

`NoSuchElementException` - if the input is exhausted

`IllegalStateException` - if this scanner is closed

reset

```
public Scanner reset()
```

Resets this scanner.

Resetting a scanner discards all of its explicit state information which may have been changed by invocations of `useDelimiter()`, `useLocale()`, or `useRadix()`.

An invocation of this method of the form `scanner.reset()` behaves in exactly the same way as the invocation

```
        scanner.useDelimiter("\\p{javaWhitespace}+")
               .useLocale(Locale.getDefault(Locale.Category.FORMAT))
               .useRadix(10);
```

Returns:

this scanner

Since:

1.6

tokens

`public Stream<String> tokens()`

Returns a stream of delimiter-separated tokens from this scanner. The stream contains the same tokens that would be returned, starting from this scanner's current state, by calling the `next()` method repeatedly until the `hasNext()` method returns false.

The resulting stream is sequential and ordered. All stream elements are non-null.

Scanning starts upon initiation of the terminal stream operation, using the current state of this scanner. Subsequent calls to any methods on this scanner other than `close()` and `ioException()` may return undefined results or may cause undefined effects on the returned stream. The returned stream's source `Spliterator` is *fail-fast* and will, on a best-effort basis, throw a `ConcurrentModificationException` if any such calls are detected during stream pipeline execution.

After stream pipeline execution completes, this scanner is left in an indeterminate state and cannot be reused.

If this scanner contains a resource that must be released, this scanner should be closed, either by calling its `close()` method, or by closing the returned stream. Closing the stream will close the underlying scanner. `IllegalStateException` is thrown if the scanner has been closed when this method is called, or if this scanner is closed during stream pipeline execution.

This method might block waiting for more input.

API Note:

For example, the following code will create a list of comma-delimited tokens from a string:

```
 List<String> result = new Scanner("abc,def,,ghi")
     .useDelimiter(",")
     .tokens()
     .collect(Collectors.toList());
```

The resulting list would contain `"abc"`, `"def"`, the empty string, and `"ghi"`.

Returns:

a sequential stream of token strings

Throws:

`IllegalStateException` - if this scanner is closed

Since:

9

findAll

`public Stream<MatchResult> findAll(Pattern pattern)`

Returns a stream of match results from this scanner. The stream contains the same results in the same order that would be returned by calling `findWithinHorizon(pattern, 0)` and then `match()` successively as long as `findWithinHorizon()` finds matches.

The resulting stream is sequential and ordered. All stream elements are non-null.

Scanning starts upon initiation of the terminal stream operation, using the current state of this scanner. Subsequent calls to any methods on this scanner other than `close()` and `ioException()` may return undefined results or may cause undefined effects on the returned stream. The returned stream's source `Spliterator` is *fail-fast* and will, on a best-effort basis, throw a `ConcurrentModificationException` if any such calls are detected during stream pipeline execution.

After stream pipeline execution completes, this scanner is left in an indeterminate state and cannot be reused.

If this scanner contains a resource that must be released, this scanner should be closed, either by calling its `close()` method, or by closing the returned stream. Closing the stream will close the underlying scanner. `IllegalStateException` is thrown if the scanner has been closed when this method is called, or if this scanner is closed during stream pipeline execution.

As with the `findWithinHorizon()` methods, this method might block waiting for additional input, and it might buffer an unbounded amount of input searching for a match.

API Note:

For example, the following code will read a file and return a list of all sequences of characters consisting of seven or more Latin capital letters:

```
 try (Scanner sc = new Scanner(Path.of("input.txt"))) {
     Pattern pat = Pattern.compile("[A-Z]{7,}");
     List<String> capWords = sc.findAll(pat)
                               .map(MatchResult::group)
```

```
                                 .collect(Collectors.toList());
 }
```

Parameters:

`pattern` - the pattern to be matched

Returns:

a sequential stream of match results

Throws:

`NullPointerException` - if pattern is null

`IllegalStateException` - if this scanner is closed

Since:

9

findAll

`public Stream<MatchResult> findAll(String patString)`

Returns a stream of match results that match the provided pattern string. The effect is equivalent to the following code:

```
     scanner.findAll(Pattern.compile(patString))
```

Parameters:

`patString` - the pattern string

Returns:

a sequential stream of match results

Throws:

`NullPointerException` - if patString is null

`IllegalStateException` - if this scanner is closed

`PatternSyntaxException` - if the regular expression's syntax is invalid

Since:

9

See Also:

`Pattern`