# Course of Programming in Java

by Łukasz Stafiniak

*Email:* lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
*Web:* www.ii.uni.wroc.pl/~lukstafi

# The Java Programming Language

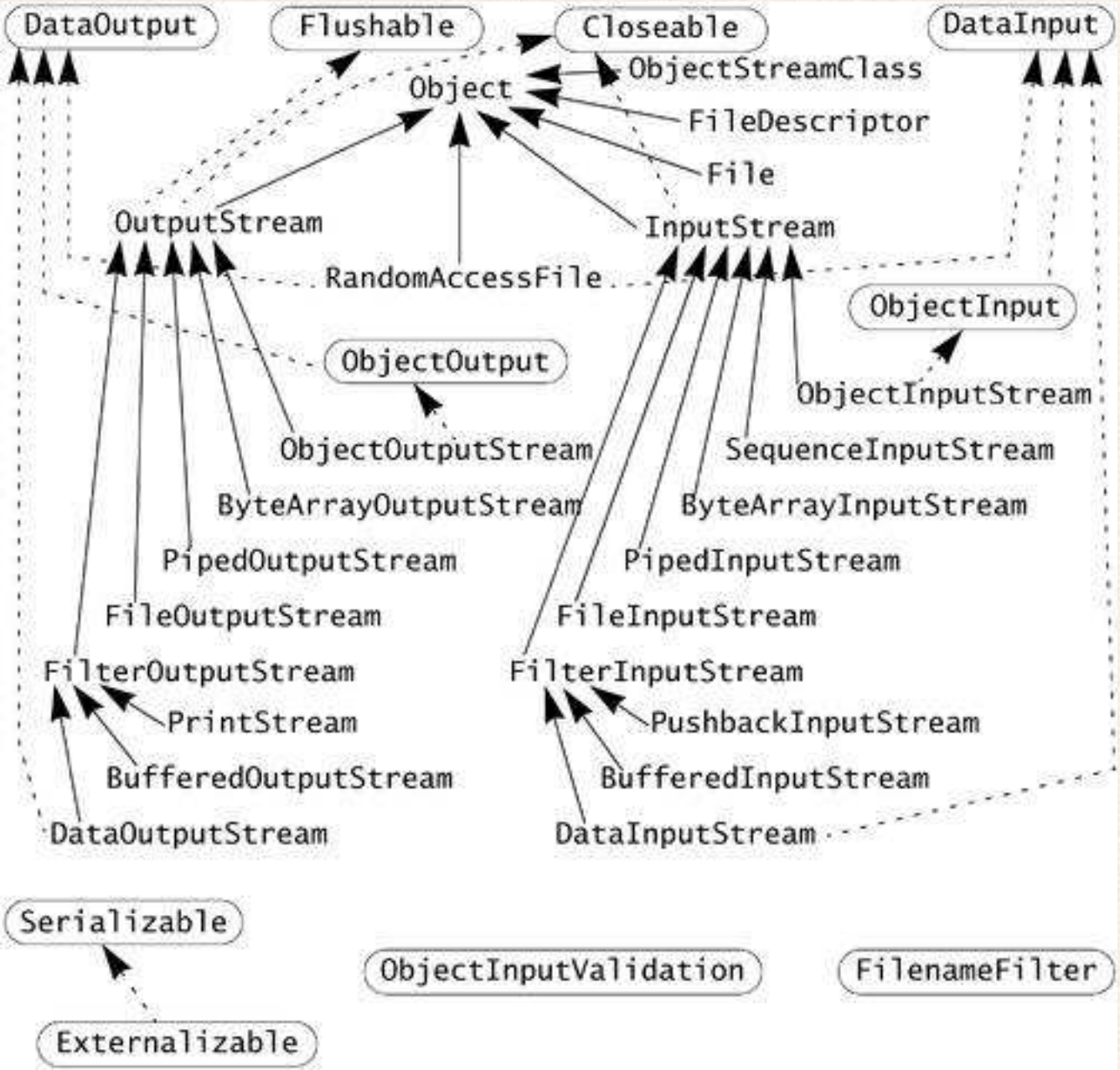
## Chapter 20. The I/O Package
## Chapter 13. Strings and Regular Expressions
## Chapter 22. Miscellaneous Utilities

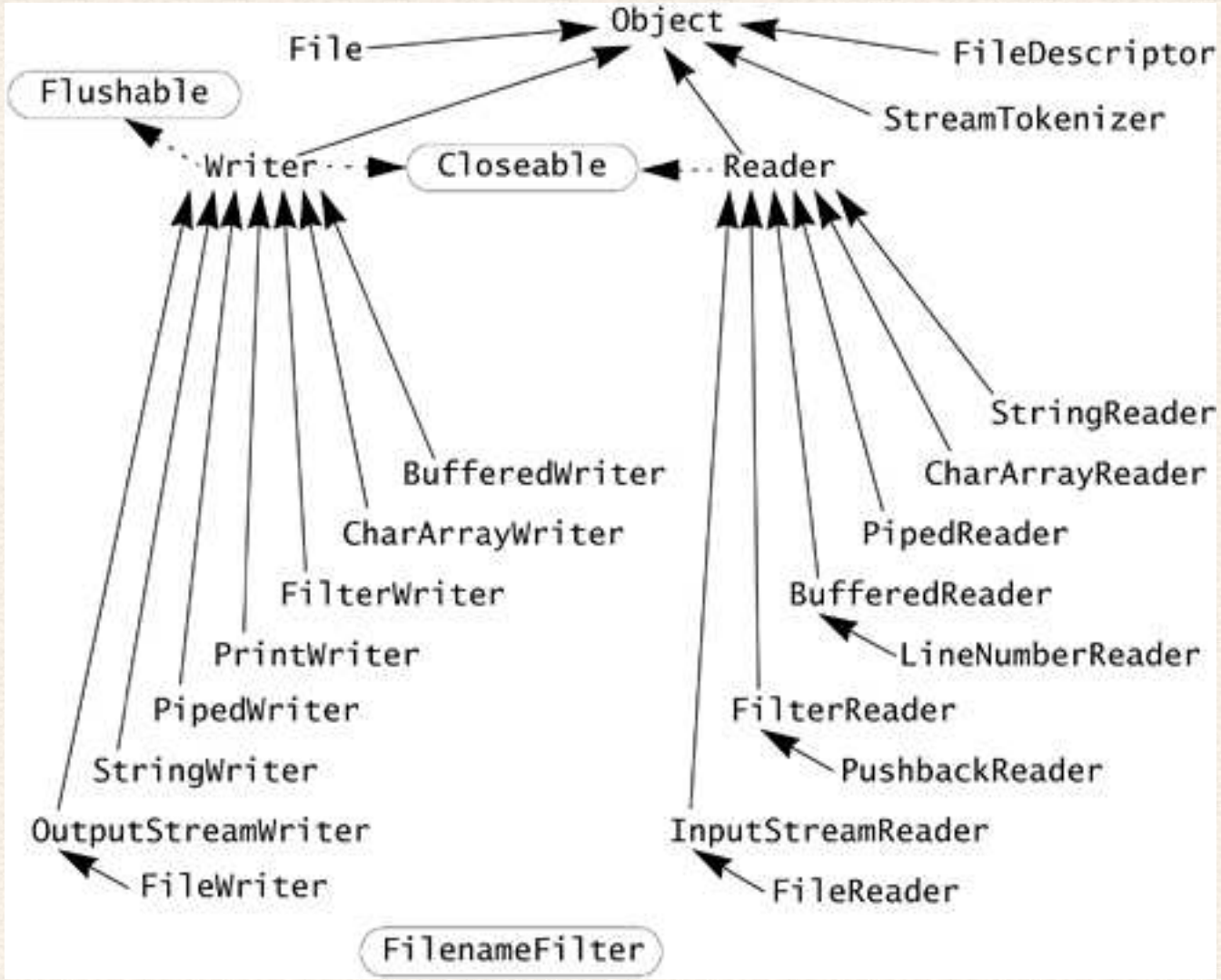
# Thinking in Java

## Chapter 19. I/O; Chapter 14. Strings

# Streams

- `java.io` is stream-based
  - `java.nio` is channel-based, non-blocking, or interruptable blocking operations
- Streams are ordered sequences of data that have a source (input streams) or destination (output streams).
- two major parts: character streams and byte streams. Characters are 16-bit `UTF-16` characters, whereas bytes are (as always) 8 bits.
- I/O is either text-based or data-based (binary).
- The byte streams are called input streams and output streams, and the character streams are called readers and writers.


DataOutput
Flushable
Closeable
DataInput
ObjectStreamClass
Object
FileDescriptor
File
OutputStream
InputStream
RandomAccessFile
ObjectInput
ObjectOutput
ObjectInputStream
ObjectOutputStream
SequenceInputStream
ByteArrayOutputStream
ByteArrayInputStream
PipedOutputStream
PipedInputStream
FileOutputStream
FileInputStream
FilterOutputStream
FilterInputStream
PrintStream
PushbackInputStream
BufferedOutputStream
BufferedInputStream
DataOutputStream
DataInputStream
Serializable
ObjectInputValidation
FilenameFilter
Externalizable


Object
File
FileDescriptor
Flushable
StreamTokenizer
Writer
Closeable
Reader
StringReader
BufferedWriter
CharArrayReader
CharArrayWriter
PipedReader
FilterWriter
BufferedReader
PrintWriter
LineNumberReader
PipedWriter
FilterReader
StringWriter
PushbackReader
OutputStreamWriter
InputStreamReader
FileWriter
FileReader
FilenameFilter

# Dealing with the File System

The `File` streams `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter` allow you to treat a file as a stream for input or output. Each type is instantiated with one of three constructors:

- A constructor that takes a `String` that is the name of the file.
- A constructor that takes a `File` object that refers to the file.
- A constructor that takes a `FileDescriptor` object.

If a file does not exist, the input streams will throw a `FileNotFoundException`.

`File` objects are created with one of four constructors:

`public File(String path)`

> Creates a `File` object to manipulate the specified `path`.

`public File(String dirName, String name)`

> Creates a `File` object for the file `name` in the directory named `dirName`.

`public File(File fileDir, String name)`

> Creates a `File` object for the file `name` in the directory named by the `File` object `fileDir`. Equiv. to `File(fileDir.getPath(), name)`.

`public File(java.net.URI uri)`

Five "get" methods retrieve information about the components of a `File` object's pathname.

```
File src = new File(".." + File.separator + "ok",
                    "FileInfo.java");
System.out.println("getName() = " + src.getName());
System.out.println("getPath() = " + src.getPath());
System.out.println("getAbsolutePath() = "
    + src.getAbsolutePath());
System.out.println("getCanonicalPath() = "
    + src.getCanonicalPath());
System.out.println("getParent() = " + src.getParent());
```

The methods `getParentFile`, `getAbsoluteFile`, and `getCanonicalFile` are analogous to `getParent`, `getAbsolutePath`, and `getCanonicalPath`, but they return `File` objects instead of strings.

Several boolean tests return information about the underlying file:

- `exists` returns `true` if the file exists in the file system.
- `canRead` returns `true` if a file exists and can be read.
- `canWrite` returns `true` if the file exists and can be written.
- `isFile` returns `true` if the file is not a directory or other special type of file.
- `isDirectory` returns `true` if the file is a directory.
- `isAbsolute` returns `true` if the path is an absolute pathname.
- `isHidden` returns `true` if the path is one normally hidden from users on the underlying system.

There are methods to inspect and manipulate the current file:

`public long lastModified()`

Returns the time the file was last modified or zero if the file does not exist.

`public long length()`

Returns the file length in bytes, or zero if the file does not exist.

`public boolean renameTo(File newName)`

Renames the file, returning `true` if the rename succeeded.

`public boolean delete()`

Deletes the file or directory (dir. must be empty) named in this `File` object, returning `true` if the deletion succeeded.

There are methods to create an underlying file or directory:

`public boolean createNewFile()`

> Creates a new empty file, named by this `File`. Returns `false` if the file already exists or cannot be created.

`public boolean mkdir()`

> Creates a directory named by this `File`, returning `true` on success.

`public boolean mkdirs()`

> Creates all directories in the path named by this `File`, returning `true` if all were created. A particular directory is created, even if it means creating other directories that don't currently exist above it.

There are methods for listing the contents of directories and finding out about root directories:

`public String[] list()`

> Lists the files in this directory. If used on something that isn't a directory, it returns `null`. Excludes the equivalent of `"."` and `".."` (the current and parent directory, respectively).

`public String[] list(FilenameFilter filter)`

> Uses `filter` to selectively list files in this directory (see `FilenameFilter` described later).

`public static File[] listRoots()`

> Returns the available filesystem roots. Windows platforms have a root directory for each active drive; `UNIX` platforms have a single / root directory.

The `FilenameFilter` interface provides objects that filter unwanted files from a list. It supports a single method:

> `boolean accept(File dir, String name)`
>
> > Returns `true` if the file named `name` in the directory `dir` should be part of the filtered output.

The `FileFilter` interface is analogous to `FilenameFilter`, but works with a single `File` object:

> `boolean accept(File pathname)`
>
> > Returns `true` if the file represented by `pathname` should be part of the filtered output.

# Some Operations on Streams

`public void flush() throws IOException`

Flushes the stream. If the stream has buffered any bytes from the various `write` methods, `flush` writes them immediately to their destination. Then, if that destination is another stream, it is also flushed.

`public void close() throws IOException`

Closes the output stream. This method should be invoked to release any resources (such as file descriptors) associated with the stream. Once a stream has been closed, further operations on the stream will throw an `IOException`. Closing a previously closed stream has no effect.

# Translating from byte to character streams

`public InputStreamReader(InputStream in)`

Read from the given `InputStream` using the default character set encoding.

`public InputStreamReader(InputStream in, String enc) throws UnsupportedEncodingException`

Read from the given `InputStream` using the named character set encoding.

`public OutputStreamWriter(OutputStream out)`

Write to the given `OutputStream` using the default character set encoding.

`public OutputStreamWriter(OutputStream out, String enc) tHRows UnsupportedEncodingException`

A character set encoding specifies how to convert between raw 8-bit "characters" and their 16-bit Unicode equivalents. Character sets are named using their standard and common names. The local platform defines which character set encodings are understood, but every implementation is required to support the following:

| | |
|---|---|
| `US-ASCII` | 7-bit `ASCII`, also known as `ISO646-US`, and as the Basic Latin block of the Unicode character set |
| `ISO-8859-1` | `ISO` Latin Alphabet No. 1, also known as `ISO-LATIN-1` |
| `UTF-8` | 8-bit Unicode Transformation Format |
| `UTF-16` | 16-bit Unicode Transformation Format, byte order specified by a mandatory initial byte-order mark (either order accepted on input, big-endian used on output) |

# Reading Lines of Text

The method `readLine` in `BufferedReader` returns a line of text as a `String`. The method `readLine` accepts any of the standard set of line separators: line feed (`\n`), carriage return (`\r`), or carriage return followed by line feed (`\r\n`). The string returned by `readLine` does not include the line separator. If the end of stream is encountered before a line separator, then the text read to that point is returned. If only the end of stream is encountered `readLine` returns `null`.

**To recall what functions for opening files, reading from files and writing to files were discussed in the class, see the code samples.**

# Object Serialization

- Serialization means converting objects from memory to byte streams
  - reconstituting an object from a byte stream is deserialization.
- `ObjectInputStream` and `ObjectOutputStream` allow you to read and write object graphs in addition to the well-known types (primitives, strings, and arrays).

```
FileOutputStream fileOut =
    new FileOutputStream("tab");
ObjectOutputStream out =
    new ObjectOutputStream(fileOut);
HashMap<?,?> hash = getHashMap();
out.writeObject(hash);
```

```
FileInputStream fileIn =
    new FileInputStream("tab");
ObjectInputStream in =
    new ObjectInputStream(fileIn);
HashMap<?,?> newHash =
    (HashMap<?,?>) in.readObject();
```

- Use `ObjectOutputStream`'s `writeUnshared` to write the object as a new distinct object, read it with `ObjectInputStream`'s `readUnshared`. Any object written into the graph by `writeUnshared` will only ever have one reference to it in the serialized data.

## Making Your Classes `Serializable`

- Implement the `Serializable` marker interface.
- Default: serializes each field that is neither `transient` nor `static`.
- Default: all serialized object fields must refer to serializable object types.
- Default: superclass either has a no-arg constructor or is `Serializable`.

```
public class Name implements java.io.Serializable {
    private String name;
    private long id;
    private transient boolean hashSet = false;
    private transient int hash;
    private static long nextID = 0;
    public Name(String name) {
        this.name = name; id = nextID++;
    }
    public int hashCode() {
        if (!hashSet) {
            hash = name.hashCode();
            hashSet = true;
        }
        return hash;
    }
}
```

## Customized Serialization

- Example: `HashMap` – default serialization is
  - wrong because hash codes may be different for deserialized entries. (For example, entries using the default `hashCode` implementation.)
  - inefficient because a hash map typically has a many empty buckets.
- Private `writeObject/readObject` methods are invoked by `ObjectOutputStream` / `ObjectInputStream`, and are responsible only for saving / loading the class's own state, including any state from non-serializable superclasses.
  - They should not invoke the superclass's `readObject` or `writeObject` method. Object serialization differs in this way from `clone` and `finalize`.

```
public class BetterName implements Serializable {
    private String name;
    private long id;
    private transient int hash;
    private static long nextID = 0;
    public BetterName(String name) {
        this.name = name; id = nextID++;
        hash = name.hashCode();
    }
    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject(); // writeObject not needed, just demo
    }
    private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        hash = name.hashCode();
    }
    public int hashCode() {
        return hash;
    }
    // ... override equals, provide other useful methods
}
```

# The `IOException` Classes

`CharConversionException extends IOException`

`EOFException extends IOException`

Thrown when the end of the file (stream) is detected while reading.

`FileNotFoundException extends IOException`

`InvalidClassException extends ObjectStreamException`

Thrown when the serialization mechanism detects a problem with a class: The serial version of the class does not match that read from the stream, the class contains unknown data types, or the class does not have an accessible no-arg constructor when needed.

`InvalidObjectException extends ObjectStreamException`

`NotActiveException extends ObjectStreamException`

> Thrown when a serialization method, such as `defaultReadObject`, is invoked when serialization is not under way on the stream.

`NotSerializableException extends ObjectStreamException`

> Thrown either by the serialization mechanism or explicitly by a class when a class cannot be serialized.

`OptionalDataException extends ObjectStreamException`

> Thrown when the optional data (not part of default serialization) in the object input stream is corrupt / not read by the reading method.

`StreamCorruptedException extends ObjectStreamException`

Thrown when internal object stream state is missing or invalid.

`UnsupportedEncodingException extends IOException`

Thrown when an unknown character encoding is specified.

`UTFDataFormatException extends IOException`

Thrown by `DataInputStream.readUTF` when the string it is reading has malformed `UTF` syntax.

# Formatter

- The primary method of a `Formatter` is `format`. In its simplest form it takes a format string followed by a sequence of arguments.
- For convenience the `PrintStream` and `PrintWriter` classes provide a `printf` method (for "print formatted") that takes the same arguments as `format` and passes them through to a `Formatter` instance.
- `%f` formats floating-point numbers, `%n` outputs endline (`"\n"` or `"\r\n"`)

  ```
  System.out.printf("The value of Math.PI is %.3f %n", Math.PI);
  ```

  which prints

  ```
  The value of Math.PI is 3.142
  ```

- The general form of a format specifier (`[...]` means optional) is

  `%[argument_index][flags][width][.precision]conversion`

| Flag | e/E | f/F | g/G | a/A | Meaning |
|---|---|---|---|---|---|
| '-' | | | | | Left justify (otherwise right justify) |
| '#' | | | x | | Always include the (hexa)decimal point |
| '+' | | | | | Always include the sign |
| ' ' | | | | | (space) Leading space for positive values |
| '0' | | | | | Use zero-padding (else spaces) |
| ',' | x | | | x | Include grouping separators |
| '(' | | | | x | Enclose negative values in parentheses |

- The `b` and `B` are boolean conversions. If the argument is `null` then the output is `"false"`; if a `boolean` then the output is either `"true"` or `"false"` depending on the argument's value; otherwise `"true"`.
- The `s` and `S` are string conversions. If the argument is `null` then the output is `"null"`. Otherwise, if the argument implements the `Formattable` interface then its `formatTo` method is invoked; otherwise, `toString` is invoked. The # flag can be passed only to a `Formattable` argument, and its effect is determined by the object.

# `String` **Utilities**

- `CharSequence` is implemented by `String`, `StringBuilder`, `StringBuffer`.
  - `char charAt(int index); int length();`

    `CharSequence subSequence(int start, int end);`

| Method | Returns Index Of... |
|---|---|
| `indexOf(int ch)` | first position of `ch` |
| `indexOf(int ch, int start)` | first position of `ch` $\geqslant$ `start` |
| `indexOf(String str)` | first position of `str` |
| `indexOf(String str, int start)` | first position of `str` $\geqslant$ `start` |
| `lastIndexOf(int ch)` | last position of `ch` |
| `lastIndexOf(int ch, int start)` | last position of `ch` $\leqslant$ `start` |
| `lastIndexOf(String str)` | last position of `str` |
| `lastIndexOf(String str, int start)` | last position of `str` $\leqslant$ `start` |

`public boolean regionMatches(boolean ignoreCase, int start, String other, int ostart, int count)` – compares substrings.

`public boolean endsWith(String suffix)` – `true` if ends with `suffix`.

```
public static String delimitedString(
    String from, char start, char end)
{
    int startPos = from.indexOf(start);
    int endPos = from.lastIndexOf(end);
    if (startPos == -1)          // no start found
        return null;
    else if (endPos == -1)       // no end found
        return from.substring(startPos);
    else if (startPos > endPos) // start after end
        return null;
    else                        // both start and end found
        return from.substring(startPos, endPos + 1);
}
```

`public String replace(char oldChar, char newChar)`

Returns a `String` with all instances of `oldChar` replaced with the character `newChar`.

`public String replace(CharSequence oldSeq, CharSquence newSeq)`

Returns a `String` with each occurrence of the subsequence `oldSeq` replaced by `newSeq`.

`public String trim()`

Returns a `String` with leading and trailing whitespace stripped.

`public String replaceFirst(String regex, String repStr)`

Returns a `String` with the first substring that matches the regular expression `regex` replaced by `repStr`. Invoked on `str`, this is equivalent to

```
Pattern.compile(regex).matcher(str).
replaceFirst(repStr)
```

`public String replaceAll(String regex, String repStr)`

Returns a `String` with all substrings that match the regular expression `regex` replaced by `repStr`. Invoked on `str`, this is equivalent to

```
Pattern.compile(regex).matcher(str).
replaceAll(repStr)
```

`public String[] split(String regex)`

Returns an array of strings resulting from splitting up this string according to the regular expression. Each match of the regular expression will cause a split in the string, with the matched part of the string removed. Trailing empty strings will be discarded. Invoked on `str`, this is equivalent to

```
Pattern.compile(regex).
split(str, limit)
```

`public String toLowerCase()`

Returns a `String` with each character converted to its lowercase equivalent.

`public String toUpperCase()`

Returns a `String` with each character converted to its uppercase equivalent.

Types that you can convert, and how to convert each to and from a `String`:

| Type | To `String` | From `String` |
|---|---|---|
| `boolean` | `String.valueOf(boolean)` | `Boolean.parseBoolean(String)` |
| `byte` | `String.valueOf(int)` | `Byte.parseByte(String, int base)` |
| `char` | `String.valueOf(char)` | `str.charAt(0)` |
| `short` | `String.valueOf(int)` | `Short.parseShort(String, int base)` |
| `int` | `String.valueOf(int)` | `Integer.parseInt(String, int base)` |
| `long` | `String.valueOf(long)` | `Long.parseLong(String, int base)` |
| `float` | `String.valueOf(float)` | `Float.parseFloat(String)` |
| `double` | `String.valueOf(double)` | `Double.parseDouble(String)` |

## `StringBuilder`

Use `StringBuilder` or `StringBuffer` to incrementally construct a string. In simple cases, the compiler optimizes `+` used on strings into `append` used on a `StringBuilder`.

# Regular Expressions

The `Pattern` class has the following methods:

> `public static Pattern compile(String regex) throws PatternSyntaxException`
>
> Compiles the regular expression into a pattern.
>
> `public static Pattern compile(String regex, int flags) throws PatternSyntaxException`
>
> The flags control how certain cases are handled.
>
> `public Matcher matcher(CharSequence input)`
>
> Will match the input against this pattern.

`public String[] split(CharSequence input)`

A convenience method that splits the given input sequence around matches of this pattern. Useful when you do not need to reuse the matcher.

`public static String quote(String str)`

Returns a string that can be used to create a pattern that would match with `str`.

The `Matcher` class has methods to match against the sequence.

`public boolean matches()`

Attempts to match the entire input sequence against the pattern.

`public boolean lookingAt()`

Attempts to match the input sequence, starting at the beginning, against the pattern, does not require that the entire input sequence be matched.

`public boolean find()`

Attempts to find the next subsequence of the input sequence that matches the pattern. If a previous invocation of `find` was successful and the matcher has not since been reset, starts at the first character not matched by the previous match.

`public boolean find(int start)`

Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index. If a match is found, subsequent invocations of the `find` method will start at the first character not matched by this match.

`public Matcher reset()`

Resets (and returns) this matcher. This discards all state and resets the append position to zero.

Once a match has been found, the following methods return more information about the match:

`public int start()`

Returns the start index of the previous match.

`public int end()`

Returns the index of the last character matched, plus one.

`public String group()`

Returns the input subsequence previously matched; the substring between `start` and `end`.

`public int groupCount()`

Returns the number of capturing groups in this matcher's pattern. Group numbers range from zero to one less than this count.

`public String group(int group)`

Returns the input subsequence matched by the given group in the previous match. Group zero is the entire matched pattern.

`public int start(int group)`

Returns the start index of the given group from the previous match.

`public int end(int group)`

Returns the index of the last character matched of the given group, plus one.

The replacement methods of `Matcher` are

`public String replaceFirst(String replacement)`

> Replaces the first occurrence of this matcher's pattern with the replacement string, returning the result. The matcher is reset before.

`public String replaceAll(String replacement)`

> Replaces all occurrences of this matcher's pattern with the replacement string, returning the result. The matcher is reset before.

`public Matcher appendReplacement(StringBuffer buf, String replacement)`

> Adds to the string buffer the characters between the current append and match positions, followed by the replacement string, and then

moves the append position to be after the match. As shown above, this can be used as part of a replacement loop. Returns this matcher.

`public StringBuffer appendTail(StringBuffer buf)`

Adds to the string buffer all characters from the current append position until the end of the sequence. Returns the buffer.

```
public static String
    swapWords(String w1, String w2, String input)
{
    String regex = "\b(" + w1 + ")(\W+)(" + w2 + ")\b";
    Pattern pat = Pattern.compile(regex);
    Matcher matcher = pat.matcher(input);
    return matcher.replaceAll("$3$2$1");
}
```

| Flag | Meaning |
|---|---|
| `CASE_INSENSITIVE` | Case-insensitive matching. By default, only handle case for the ASCII characters. |
| `UNICODE_CASE` | Unicode-aware case folding when combined with `CASE_INSENSITIVE` |
| `CANON_EQ` | Canonical equivalence. If a character has multiple expressions, treat them as equivalent. For example, å is canonically equivalent to `a\u030A`. |
| `DOTALL` | Dot-all mode, where `.` matches line breaks, which it otherwise does not. |
| `MULTILINE` | Multiline mode, where `^` and `$` match at lines embedded in the sequence, not just at the start end of the entire sequence |
| `UNIX_LINES` | Unix lines mode, where only `\n` is considered a line terminator. |
| `COMMENTS` | Comments and whitespace in pattern. Whitespace will be ignored, and comments starting with # are ignored up to the next end of line. |
| `LITERAL` | Enable literal parsing of the pattern |

# Scanner

- `Scanner` can be created from any `Readable`, which includes any `Reader`
- `Scanner` can read a number in any format `Formatter` can use
- `Scanner` implements `Iterator<String>`
  - but not `Iterable` so it cannot be used with the `for` loop.
- Besides `hasNext()` and `next()`, also `hasNextType` and `nextType` returning the primitive Type, e.g. `hasNextDouble()`
- `hasNext` and `next` can also take `String` or `Pattern` to match against regular expression
- If there is no next token then `NoSuchElementException` is thrown. If the method requires a specific type or pattern of token, and the next token does not match, then `InputMismatchException` is thrown.
- A scanner identifies tokens by looking for a delimiter pattern (by default whitespace) in the input stream of characters. You can set the delimiter to a different pattern with the `useDelimiter` method.

`public String findInLine(Pattern pattern)`

> Attempts to find the given pattern before the next line separator is encountered. The delimiters of the scanner are ignored during this search. The scanner advances to the next position after the matching input, or returns `null`.

- `public String findWithinHorizon(Pattern/String pattern, int horizon)` finds `pattern` (within `horizon` characters) ingoring scanner delimiters, e.g. `findWithinHorizon(pattern, 0)` can find a multi-word pattern (0 means without horizon).
- `hasNextLine and nextLine` methods tokenize input a complete line at a time.
- The `skip` method can be used to skip over the input that matches a given pattern. (Returns the scanner.)

```
static final int CELLS = 4;
public static List<String[]> readCSVTable(Readable source)
    throws IOException {
    Scanner in = new Scanner(source);
    List<String[]> vals = new ArrayList<String[]>();
    String exp = "^(.*),(.*),(.*),(.*)";
    Pattern pat = Pattern.compile(exp, Pattern.MULTILINE);
    while (in.hasNextLine()) {
        String line = in.findInLine(pat);
        if (line != null) {
            String[] cells = new String[CELLS];
            MatchResult match = in.match();
            for (int i = 0; i < CELLS; i++)
                cells[i] = match.group(i+1);
            vals.add(cells);
            in.nextLine();  // skip newline
        }
        else throw new IOException("input format error");
    }
    IOException ex = in.ioException();
    if (ex!= null) throw ex;
    return vals;
}
```

```
Scanner in = new Scanner(source);
Pattern COMMENT = Pattern.compile("#.*");
String comment;
// ...
while (in.hasNext()) {
    if (in.hasNext(COMMENT)) {
        comment = in.findWithinHorizon(COMMENT, 0);
        in.nextLine();
    }
    else {
        // process other tokens
    }
}
```

Regular expressions *? and +? are non-greedy qualifiers: they get the fewest possible characters.

```
public static void printAttrs(Writer dest, Attr[] attrs) {
    PrintWriter out = new PrintWriter(dest);
    out.printf("%d attrs%n", attrs.length);
    for (int i = 0; i < attrs.length; i++) {
        Attr attr = attrs[i];
        out.printf("%s=%s%n",
                   attr.getName(), attr.getValue());
    }
    out.flush();
}
public static Attr[] scanAttrs(Reader source) {
    Scanner in = new Scanner(source);
    int count = in.nextInt();
    in.nextLine();    // skip rest of line
    Attr[] attrs = new Attr[count];
    Pattern attrPat =
        Pattern.compile("(.*?)=(.*)$", Pattern.MULTILINE);
    for (int i = 0; i < count; i++) {
        in.findInLine(attrPat);
        MatchResult m = in.match();
        attrs[i] = new Attr(m.group(1), m.group(2));
    }
    return attrs;
}
```