HBO Graduaat Informatica Optie Programmeren

Java In Depth
Java I/O

cvo leerstad

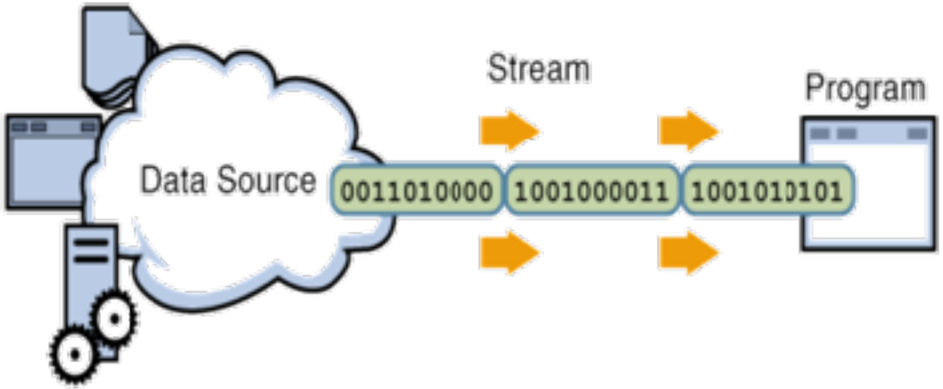
volwassenenonderwijs

Streams

- What is a Stream?
  - Sequence of bytes or characters for reading information of a source ***B*** from a source ***A***

- `java.IO.InputStream`
  - `int read ()`
  - `int read (byte[] b)`
  - `int read (byte[] b, int off, int len)`

- `java.IO.Reader`
  - `int read ()`
  - `int read (char[] cbuf)`
  - `Int read (char[] cbuf , int off, int len)`

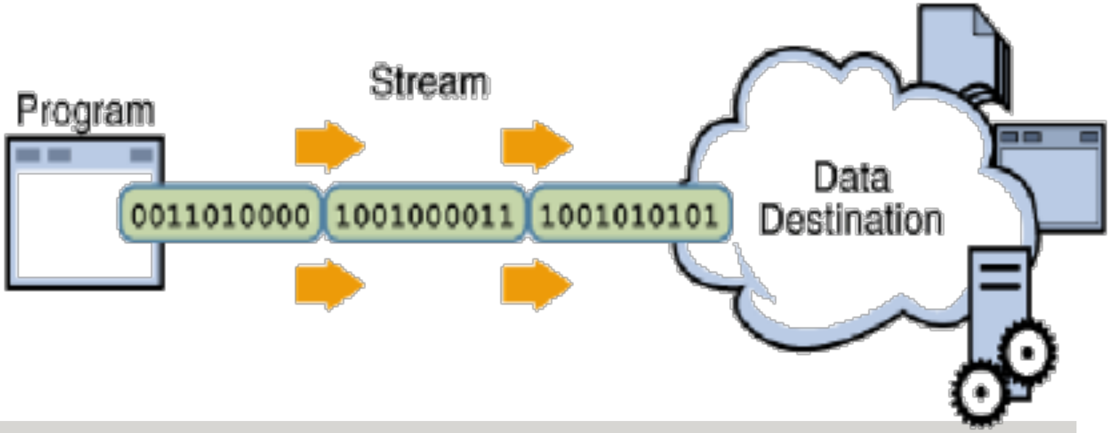


Streams

- What is a Stream?
  - Sequence of bytes or characters for writing information from a source ***A*** to a source ***B***

- `java.io.OutputStream`
  - `int write (int c)`
  - `int write (byte[] b)`
  - `int write (byte[] b, int off, int len)`

- `java.io.Writer`
  - `int write (int c)`
  - `int write (char[] cbuf)`
  - `int write (char[] cbuf , int off, int len)`



Reading and writing operations

- Reading operation
  - Open the stream
  - While more information, read information
  - Close the stream (try-with-resources Statement)

    Use `java.io.InputStream` / java.io.Reader objects

- Writing operation
  - Open the stream
  - While more information, write information
  - Close the stream (try-with-resources Statement)

    Use java.io.OutputStream / java.io.Writer objects

The try-with-resources Statement

```
public String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                    new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

- New superinterface **`java.lang.AutoCloseable`**
- All **`AutoCloseable`** (throws Exception) and by extension **`java.io.Closeable`** (throws **`IOException`**) types useable with try-with-resources
- Anything with a **`void close()`** method is a candidate
- JDBC 4.1 retrofitted as **`AutoCloseable`** too

Streams

- Most common places to write/to read data are :
  - a sequential file
  - a String
  - a pipe
  - the system console
  - an array of characters
  - a URL for an HTTP GET/POST
  - a random access file
  - an array of bytes
  - a socket

Streams types and hierarchies

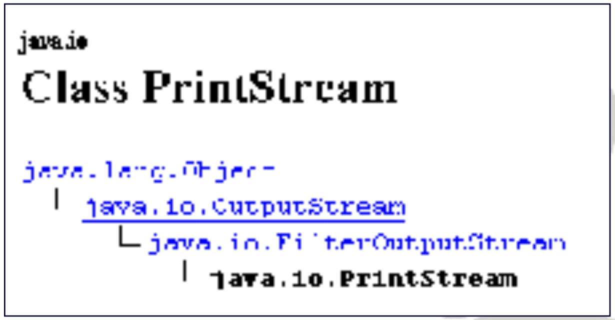
- Streams types
  - ***Data sink streams:*** read and write from/to specialized data sink
    - Memory streams: characters chains manipulation
    - File streams :read and write from/to files
    - Pipe streams :exchange information between threads
  - ***Processing streams:*** streams with additional functionalities
    - Encoding, buffering, filtering, …

- Streams hierarchies
  - ***Byte streams:*** writes bytes (8/32 bits) …………………………
    - java.io.InputStream / java.io.OutputStream
  - ***Character streams:*** writes ***UNICODE*** characters (16/32 bits)
    - java.io.Reader / java.io.Writer

Standard I/O streams

- Standard output
  - ***java.io.PrintStream System.out***
    - processing stream which writes information into a console
    - generally used to display *information* into the console at runtime
  - ***java.io.PrintStream System.err***
    - processing stream which writes information into a console
    - generally used to display *error messages* in the console at runtime

- Main methods
  - void print (*object, primitives, array*)
    - calls toString () on objects
  - void println (*object, primitives, array*)
    - calls toString () on objects
    - appends line return



java.io

Class PrintStream

```
java.lang.Object
  |
  +--java.io.OutputStream
        |
        +--java.io.FilterOutputStream
              |
              +--java.io.PrintStream
```

Standard I/O streams

- Standard input
  - ***java.io.inputStream.System.in***
    - processing stream which reads input entered by the user (console)
    - blocks the program flow, waiting for user input
- Main methods
  - int read ()
    - reads following byte of the stream
    - returns an integer between 0 and 255 or -1 if no more bytes
  - int read (byte[] b)
    - reads the bytes of a byte[]
    - returns the number of bytes read or -1 if no more bytes
  - int read (byte[] b, int offset, int length)
    - reads partially a byte[]
    - offset represent the position of the first byte to read, length the number of bytes to read
    - returns the number of bytes read or -1 if no more bytes

Standard I/O streams

- Standard input implementation

```
System.out.println ("Do you like JAVA ?");
System.out.println ("Y | N | E");
int c;
while ( (c = System.in.read() ) != -1) {
    if ( (char) c = = 'Y' | | (char) c = = 'y')
        System.out.println ("You have got the right language!");
    if ( (char) c = = 'N' | | (char) c = = 'n')
        System.err.println ("I don't understand, try again!");
    if ( (char) c = = 'E' | | (char) c = = 'e') {
        System.exit (0);
}
```

Standard I/O streams

- Redirect standard I/O streams
  - Redirect the standard input/output using System class methods:
    - `static void setOut (PrintStream out)`
    - `static void setErr (PrintStream out)`
    - `static void setIn (InputStream in)`

Portability of I/O

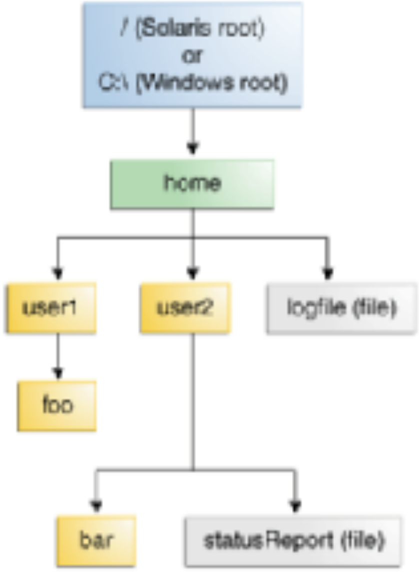
- The basic portability approach of the Java runtime library is to have the same method do slightly different things appropriate to each platform

  For example :

  - `java.io.File.separator`
    - The system-dependent default name-separator character, represented as a string for convenience.
  - `java.io.File.pathSeparator`
    - The system-dependent path-separator character, represented as a string for convenience

Path

- A path is either relative or absolute. An absolute path always contains the root element and the complete directory list required to locate the file. For example, /home/peter/statusReport is an absolute path. All of the information needed to locate the file is contained in the path string.
- A relative path needs to be combined with another path in order to access a file. For example, joe/foo is a relative path. Without more information, a program cannot reliably locate the joe/foo directory in the file system.



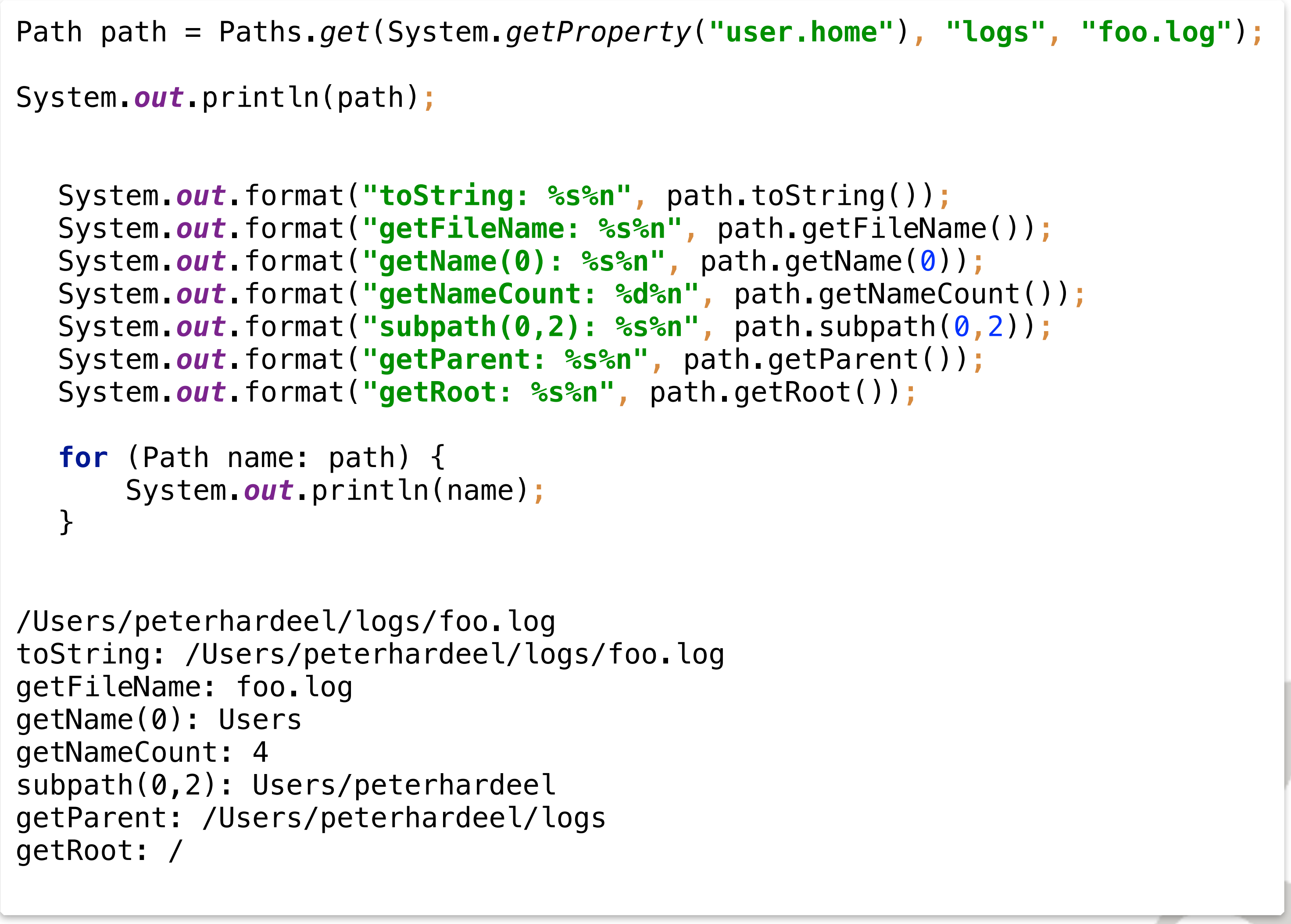
c:\home\peter\statusreport

/home/peter/statusReport

Path

- The `Path` class is a programmatic representation of a path in the file system. A Path object contains the file name and directory list used to construct the path, and is used to examine, locate, and manipulate files.

Path



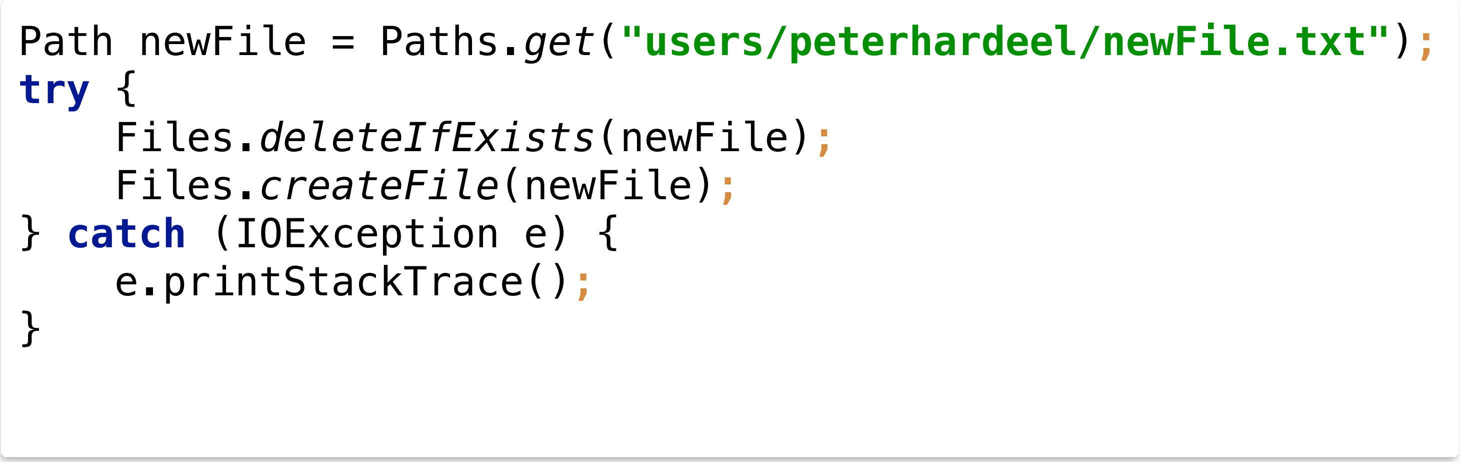
```
Path path = Paths.get(System.getProperty("user.home"), "logs", "foo.log");

System.out.println(path);


  System.out.format("toString: %s%n", path.toString());
  System.out.format("getFileName: %s%n", path.getFileName());
  System.out.format("getName(0): %s%n", path.getName(0));
  System.out.format("getNameCount: %d%n", path.getNameCount());
  System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
  System.out.format("getParent: %s%n", path.getParent());
  System.out.format("getRoot: %s%n", path.getRoot());

  for (Path name: path) {
      System.out.println(name);
  }


/Users/peterhardeel/logs/foo.log
toString: /Users/peterhardeel/logs/foo.log
getFileName: foo.log
getName(0): Users
getNameCount: 4
subpath(0,2): Users/peterhardeel
getParent: /Users/peterhardeel/logs
getRoot: /
```

Files

- **`java.nio.file.Files`** is a very powerful class that provides static methods for handling files and directories as well as reading from and writing to a file. With it you can create and delete a path, copy files, check if a path exists, and so on. In addition, Files comes with methods for creating stream objects that you'll find useful when working with input and output streams.

Files

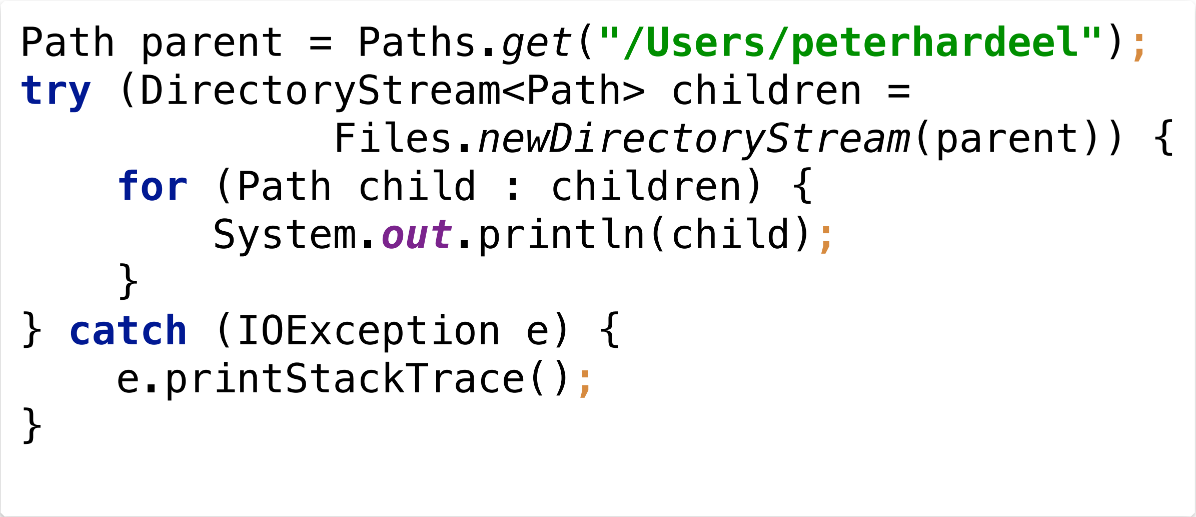
- Creating and Deleting Files and Directories



```
Path newFile = Paths.get("users/peterhardeel/newFile.txt");
try {
    Files.deleteIfExists(newFile);
    Files.createFile(newFile);
} catch (IOException e) {
    e.printStackTrace();
}
```

Files

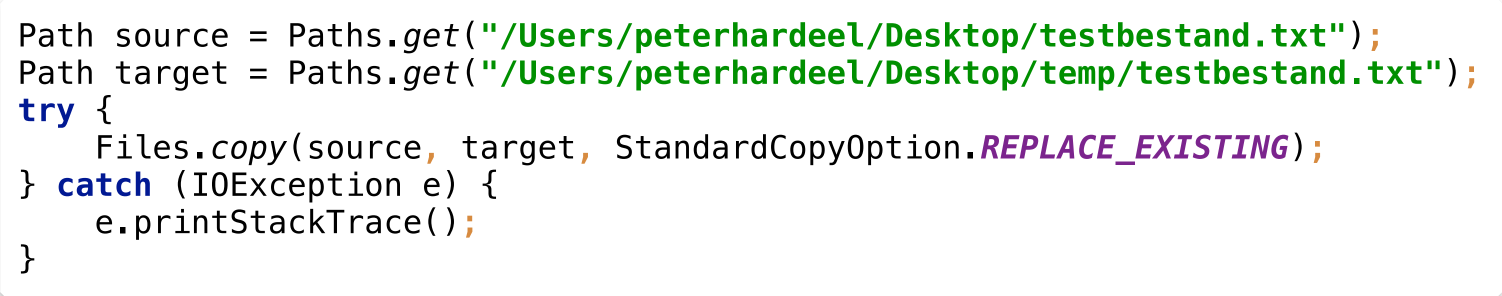
- Retrieving a Directory's Objects



```
Path parent = Paths.get("/Users/peterhardeel");
try (DirectoryStream<Path> children =
             Files.newDirectoryStream(parent)) {
    for (Path child : children) {
        System.out.println(child);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Copy and Moving Files

- There are three copy/move methods for copying/moving files and directories. The easiest one to use is this one.



```
Path source = Paths.get("/Users/peterhardeel/Desktop/testbestand.txt");
Path target = Paths.get("/Users/peterhardeel/Desktop/temp/testbestand.txt");
try {
    Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) {
    e.printStackTrace();
}
```

- ATOMIC_MOVE. Move the file as an atomic file system operation.
- COPY_ATTRIBUTES. Copy attributes to the new file.
- REPLACE_EXISTING. Replace an existing file if it exists.

Reading and writing to a File

- The Files class provides methods for reading from and writing to a small binary and text file. The readAllBytes and readAllLines methods are for reading from a binary and text file, respectively.

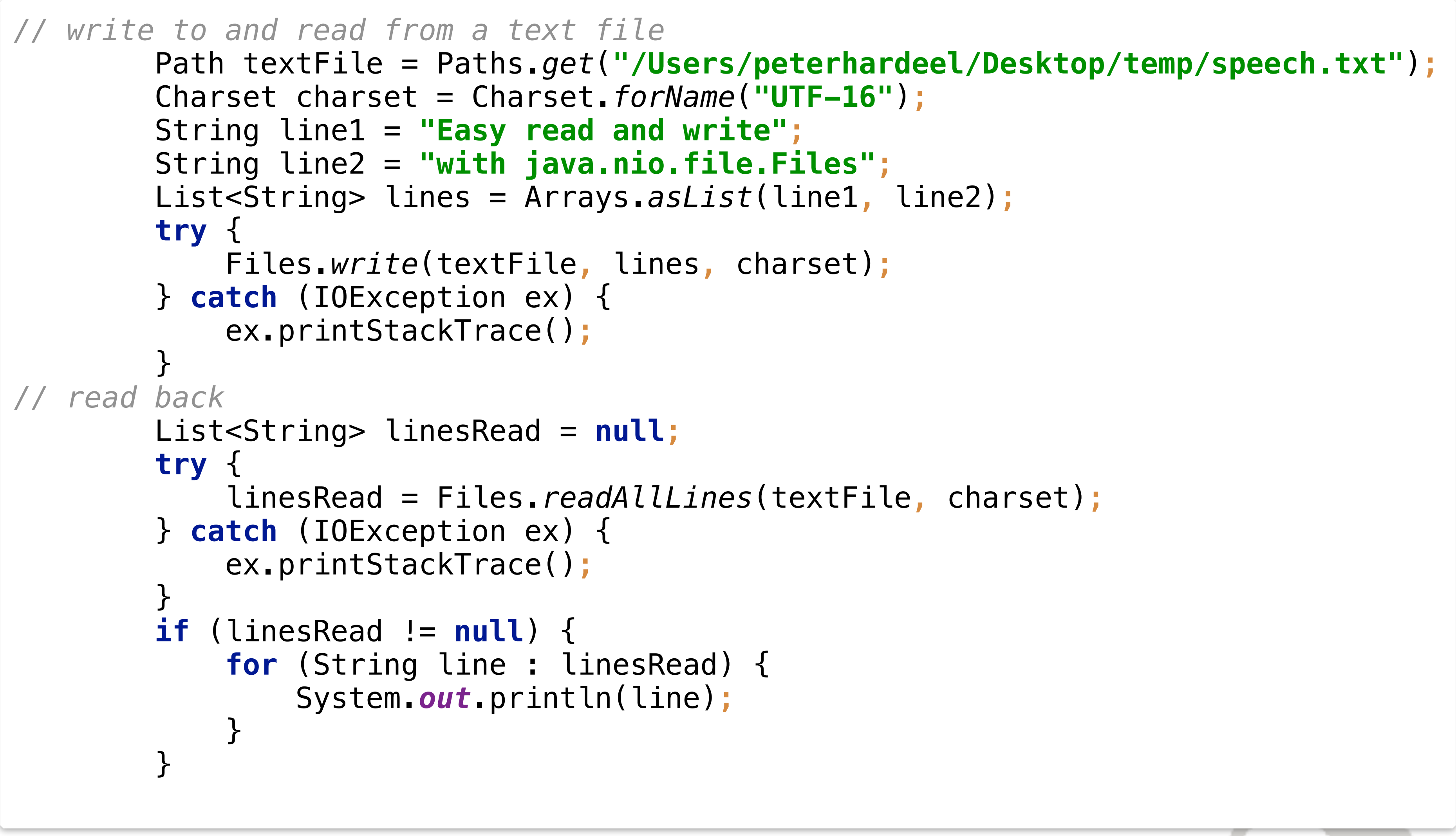
```
public static byte[] readAllBytes(Path path) throws java.io.IOException

public static List<String> readAllLines(Path path,
java.nio.charset.Charset charset ) throws java.io.IOException
```

Reading and writing to a File

- These write methods are for writing to a binary and text file, respectively.

```
public static Path write(Path path, byte[] bytes, OpenOption... options)
throws java.io.IOException

public static Path write(Path path, java.lang.Iterable<? extends
CharSequence> lines,java.nio.charset.Charset charset, OpenOption...
options) throws java.io.IOException
```

Reading-Writing example



```java
// write to and read from a text file
        Path textFile = Paths.get("/Users/peterhardeel/Desktop/temp/speech.txt");
        Charset charset = Charset.forName("UTF-16");
        String line1 = "Easy read and write";
        String line2 = "with java.nio.file.Files";
        List<String> lines = Arrays.asList(line1, line2);
        try {
            Files.write(textFile, lines, charset);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
// read back
        List<String> linesRead = null;
        try {
            linesRead = Files.readAllLines(textFile, charset);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        if (linesRead != null) {
            for (String line : linesRead) {
                System.out.println(line);
            }
        }
```

Wrappers

What is a wrapper ?

- Wrapping an object means accessing its features through some other object
- The wrapper object will augment/improve the features available from the first object
- Wrappers are written with a constructor whose argument is the object they will wrap

Wrappers

```
NicePrinter wrappedObject = new NicePrinter();
FrenchToEnglish wrapper = new
FrenchToEnglish(wrappedObject);
```

- Wrappers are widely used in Java I/O library

Wrappers (2)

- You can layer several wrappers on top of each other
  - you deal only with the outermost one
  - each wrapper does its own special value-add
  - each wrapper send its data on to the next one

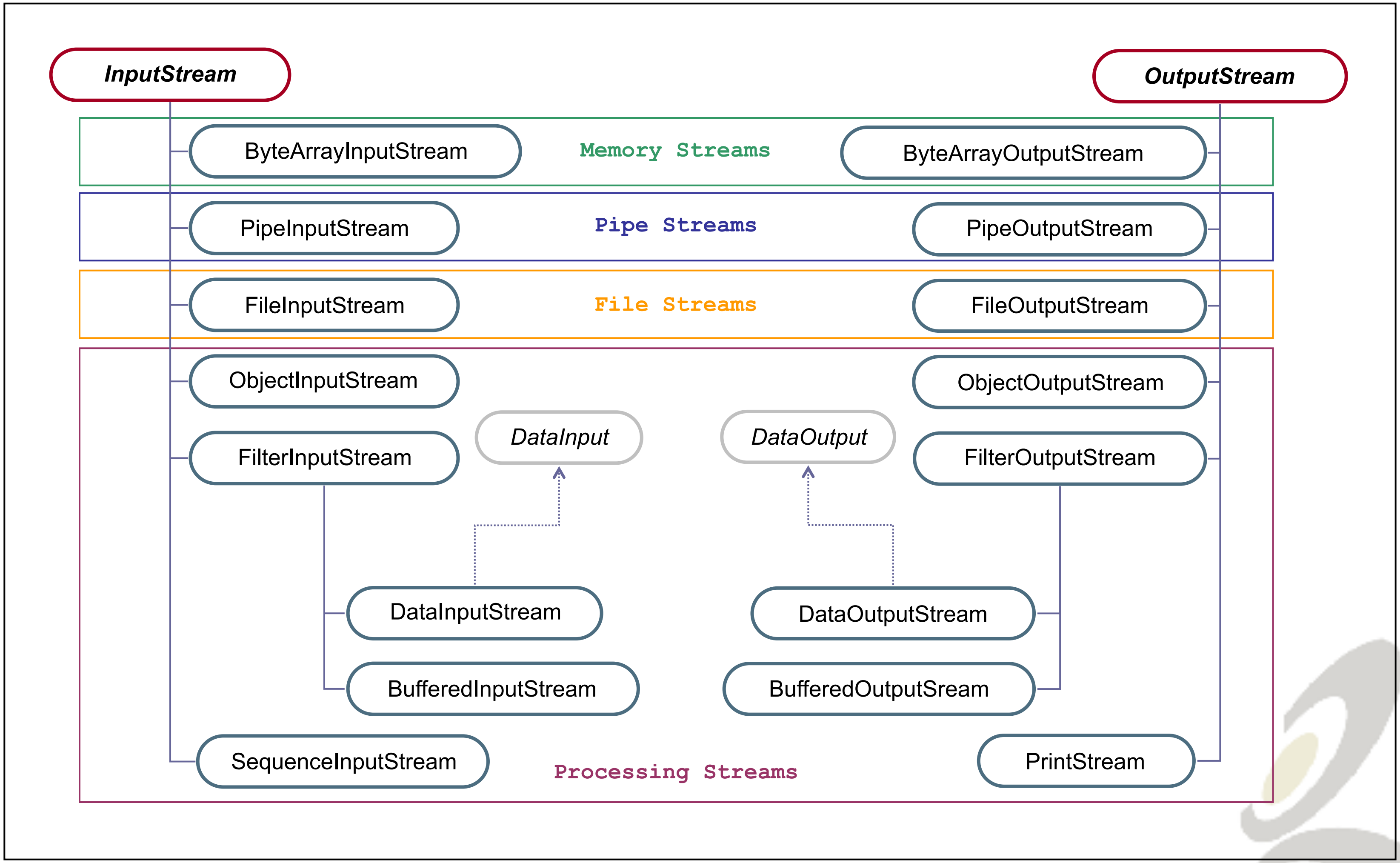
- Wrapping is a design pattern a.k.a. the “Decorator” pattern

Readers, Writers and Streams

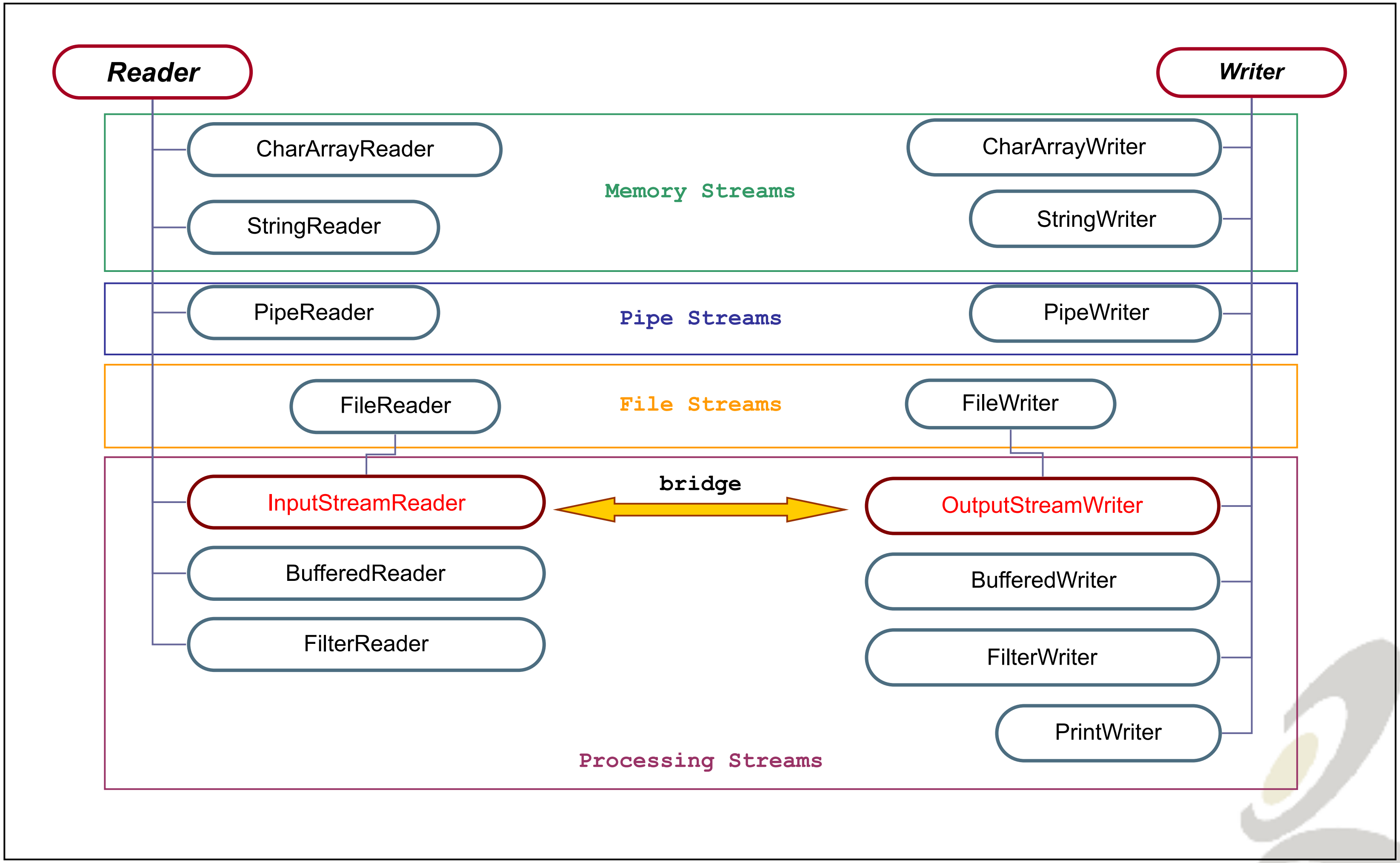
- Readers and Writers (*character* streams)
  - Reader classes are able to get Unicode character input *two bytes* at a time
  - Writer classes are able to do Unicode character output *two bytes* at a time

- Byte Streams
  - Input and output streams operate on data *one byte* at a time
    - Used when you want to read ASCII or binary data

Input / OutputStream hierarchy



Reader / Writer hierarchy



Reading (1)

- Reader and InputStream define similar APIs but for different data types.
  - `Reader:`
    - int read()
    - int read(char cbuf[])
    - int read(char cbuf[], int offset, int length)
  - `InputStream:`
    - int read()
    - int read(byte cbuf[])
    - int read(byte cbuf[], int offset, int length)

Reading (2)

- Also, both `Reader` and `InputStream` provide methods for
  - marking a location in the stream
    - `void mark(int readAheadLimit)`
  - skipping input
    - `long skip(long n)`
  - resetting the current position
    - `void reset()`

Writing

- `Writer` and `OutputStream` are similarly parallel.
  - `Writer`:
    - int write(int c)
    - int write(char cbuf[])
    - int write(char cbuf[], int offset, int length)
  - `OutputStream`:
    - int write(int c)
    - int write(byte cbuf[])
    - int write(byte cbuf[], int offset, int length)

Open and closing streams

- All of the streams (readers, writers, input and output streams) are automatically opened when created.
- You can close any stream explicitly by calling its **close()** method. Since most stream classes now implement java.lang.AutoCloseable, you can create a stream in a try-with-resources statement and get the streams automatically closed for you.
- Garbage Collector can implicitly close the stream, which occurs when the object is no longer referenced.

Writers revisited

- Use a `writer` when you want to output printable, internationalizable 16-bit characters.
- Use `FileWriter`, `CharArrayWriter`, `PipedWriter` or `StringWriter` depending on where you want the chars to go.
- Optionally wrap on of the writers in either (or both) a `BufferedWriter` or your subclass of a `FilterWriter`.

Writers revisited

- Wrap a `PrintWriter` on top, and use its print methods to do the output.

Readers revisited

- Use a `reader` when you want to input printable, internationalizable 16-bit characters.
- Use `FileReader`, `CharArrayReader`, `PipedReader` or `StringReader` depending on where the chars come from.
- Optionally wrap on of the readers in either (or both) a `BufferedReader` or your subclass of a `FilterReader`.

Output Streams revisited

- Use an `OutputStream` when you want to output ASCII text or binary values.
- Choose a `FileOutputStream` or one of the `getOutputStream()` methods, depending on where you want the chars to go.
- Optionally wrap in an arbitrary number of `OutputStream` filters, buffers, compressors, encoders, etc.
- Wrap a `DataOutputStream` on top, and use its write methods to output numbers binary.

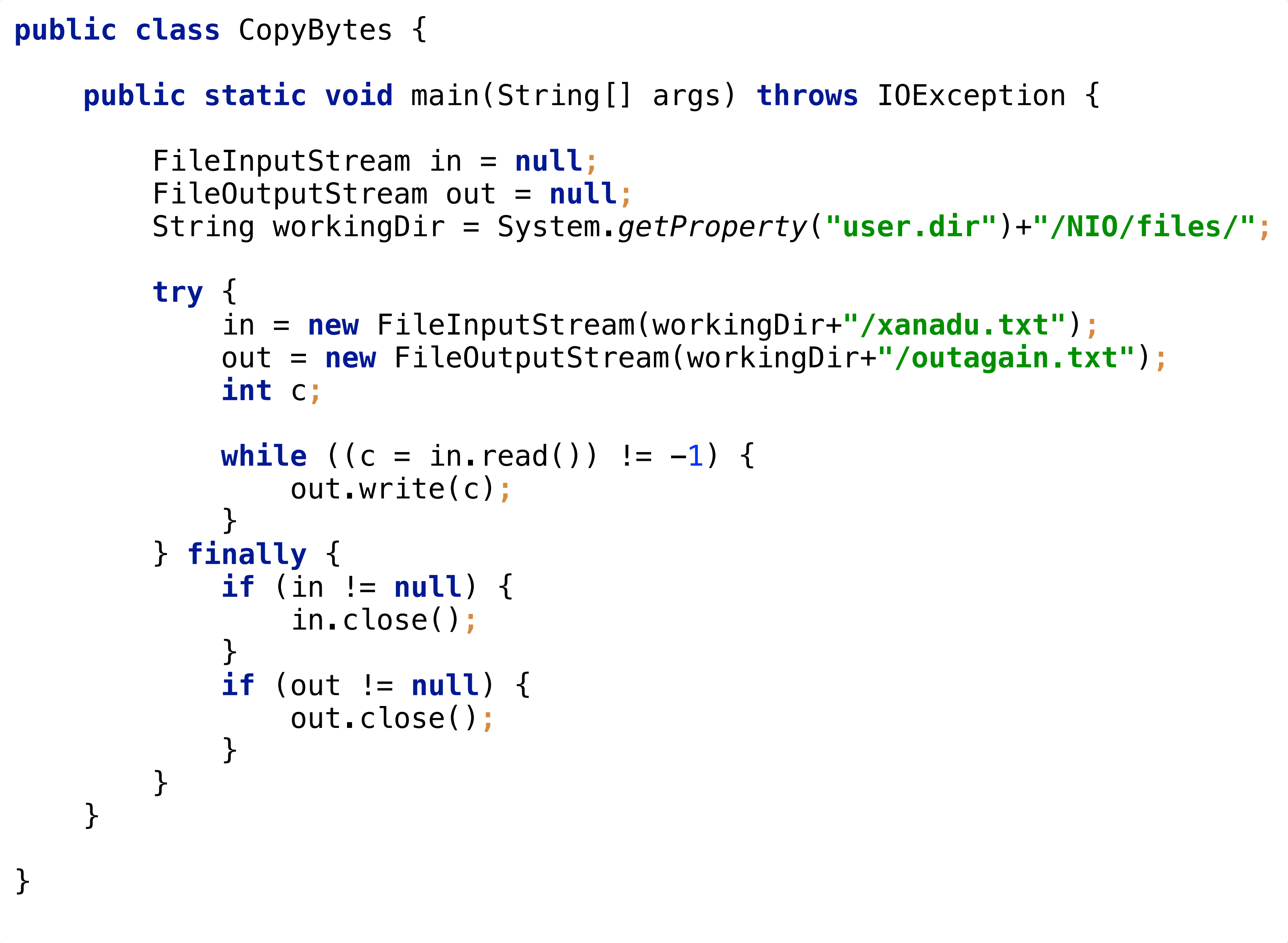
Input Streams revisited

- Use an Input Stream when you want to input ASCII text or binary values.
- Choose a `FileInputStream` or one of the `getInputStream()` methods, depending on where you want the bytes to come from.

Input Streams revisited

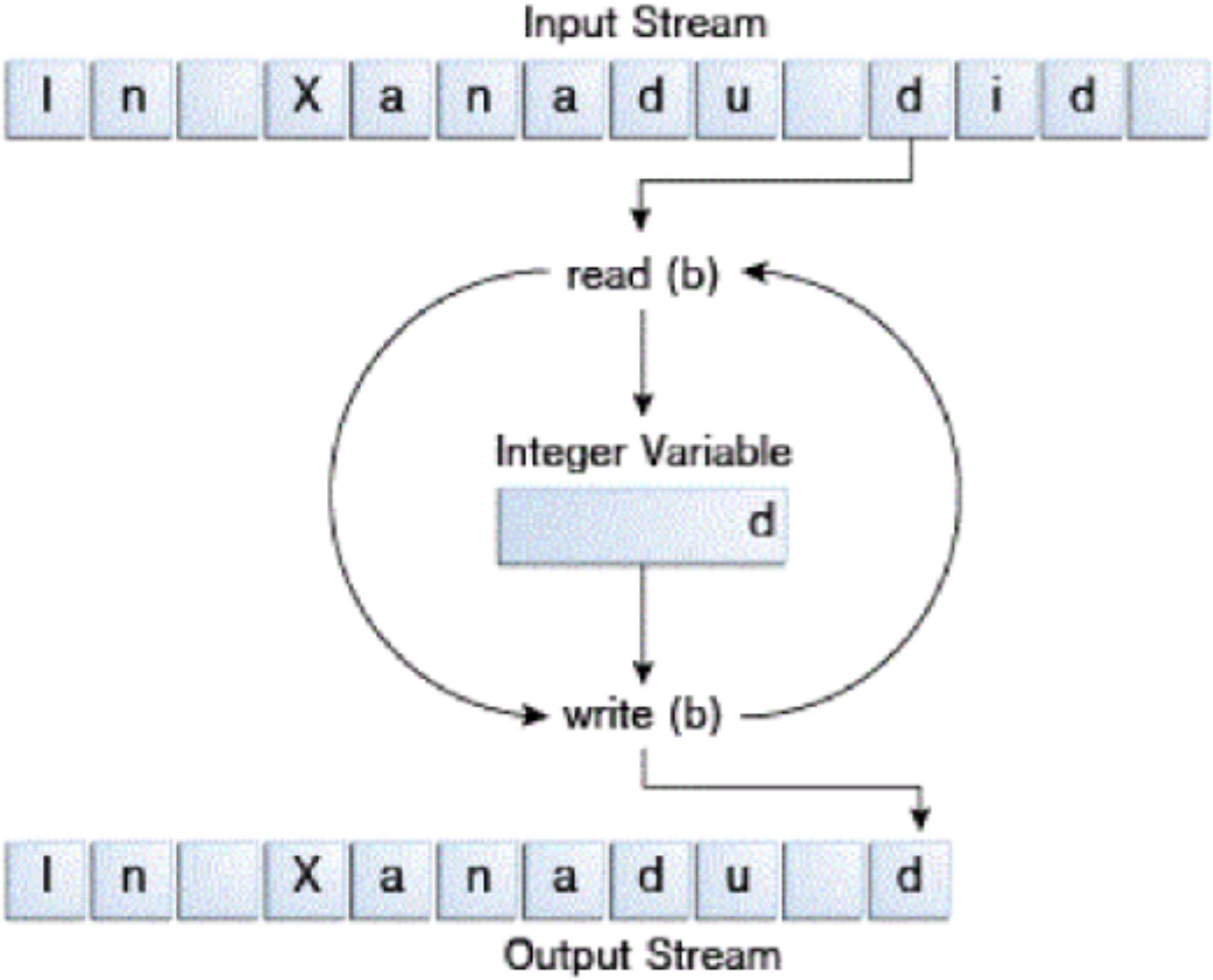
- Optionally wrap in an arbitrary number of `InputStream` filters, buffers, expanders, decoders, etc.
- Wrap a `DataInputStream` on top, and use its read methods to do the input (use `ObjectInputStream` if you are reconstituting objects rather than reading data).
- If using a buffer, it should directly wrap the `FileInputStream` (so that as much as possible of the pipeline of classes if buffered)

FileInputStream & FileOutputStream



```
public class CopyBytes {

    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;
        String workingDir = System.getProperty("user.dir")+"/NIO/files/";

        try {
            in = new FileInputStream(workingDir+"/xanadu.txt");
            out = new FileOutputStream(workingDir+"/outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }

}
```

FileInputStream & FileOutputStream



FileReader & FileWriter

```java
public class CopyCharacters {

    public static void main(String[] args) throws IOException {

        FileReader inputStream = null;
        FileWriter outputStream = null;
        String workingDir = System.getProperty("user.dir") + "/NIO/files/";

        try {
            inputStream = new FileReader(workingDir + "/xanadu.txt");
            outputStream = new FileWriter(workingDir + "/characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Detailed Overview

The following slides will show you how to

- Output Double Byte characters
- Input Double Byte characters
- Output ASCII and binary values
- Input ASCII and binary values

Output Double Byte characters

- Send output to
  - a file, then use ***FileWriter***
  - a char array, then use ***CharArrayWriter***
  - a String, then use ***StringWriter***
  - a pipe to be read by *PipedReader* in another thread, then use ***PipedWriter***

Remember the Java API ?
Check constructors for above classes in java.io.* package !

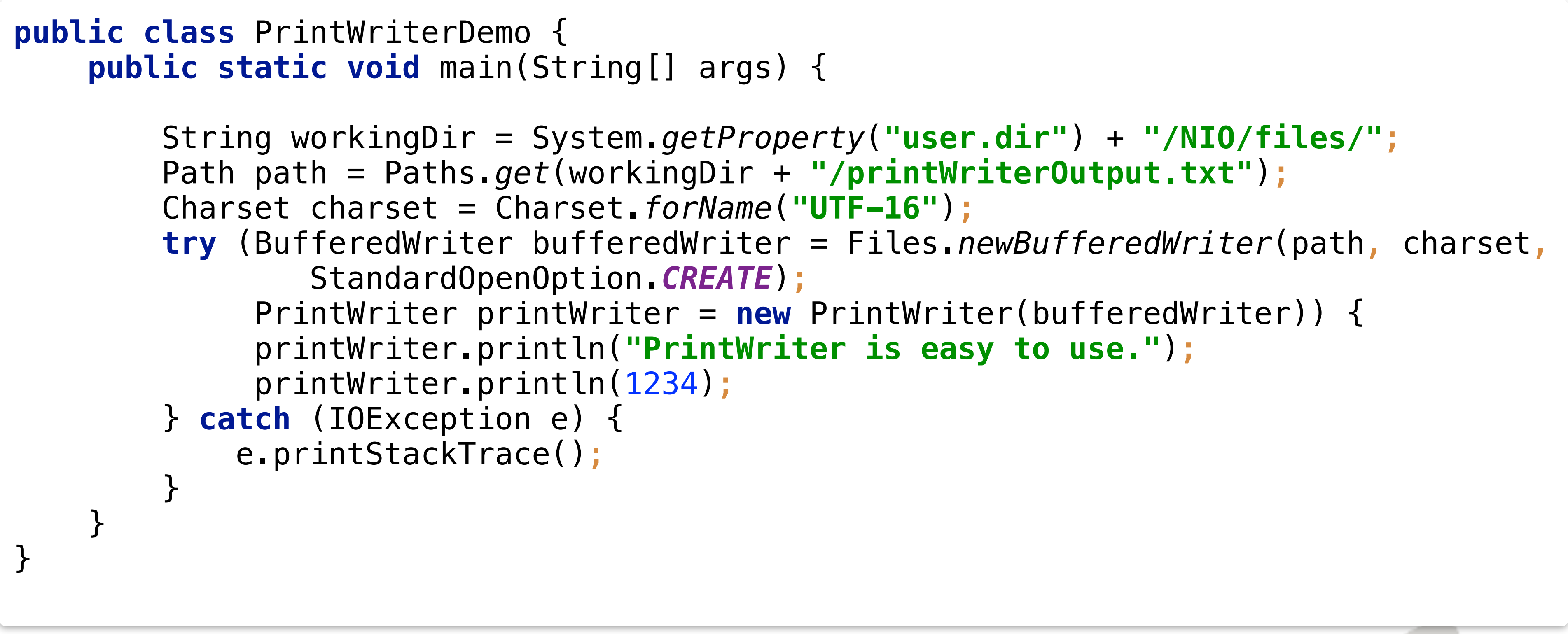
java.io.PrintWriter

- Actually transfers data to the Writer destination as printable strings
- A `print()` method for most primitive types

  (Not for byte, because byte oriented output is done with an OutputStream)

- `print()` methods are all implemented calling the write() methods
- A `println()` method that follows the output with the end of line sequence for that platform

Using PrintWriter



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

        String workingDir = System.getProperty("user.dir") + "/NIO/files/";
        Path path = Paths.get(workingDir + "/printWriterOutput.txt");
        Charset charset = Charset.forName("UTF-16");
        try (BufferedWriter bufferedWriter = Files.newBufferedWriter(path, charset,
                StandardOpenOption.CREATE);
             PrintWriter printWriter = new PrintWriter(bufferedWriter)) {
             printWriter.println("PrintWriter is easy to use.");
             printWriter.println(1234);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Input Double Byte characters

(Similar to output)

Get input from

- a file, then use ***FileReader***
- a char array, then use ***CharArrayReader***
- a String, then use ***StringReader***
- a pipe written by a ***PipedWriter*** in another thread, then use ***PipedReader***

Reader Wrappers

- ***BufferedReader***
  - provides performance boost
  - has a `readLine()` method
- ***FilterReader***
  - subclass the FilterReader, override needed methods

Reader Wrappers

- LineNumberReader
  - keeps track of line number count on stream
  - has a `getLineNumber()` method
- PushbackReader
  - maintains an internal buffer that allows characters to be 'pushed' back into the stream after they haven been read, allowing the next read to read them again

Output ASCII and binary values

- Send binary output to
  - a file, then use `FileOutputStream`
  - a byte array, then use `ByteArrayOutputStream`
  - a pipe to be read by `PipedInputStream` in another thread, then use `PipedOutputStream`
  - a String, don't use a Stream class, use `StringWriter`

Output Stream Wrappers (1)

- `java.io.DataOutputStream`
  - use when output is in binary format, used for later processing by another program
  - writeXXX() methods for all primitives types and Strings
  - depending on the used write method, strings will be written as :
    - 16-bit Unicode chars
    - 8-bit bytes discarding high-order byte of each char
    - UTF-encoded format where chars are 1-3 bytes

Output Stream Wrappers (2)

- `java.io.PrintStream`
  - use when the output is
    - ISO8859-1 readable text and numbers, but not needing internationalization
    - ASCII bytes
    - Internationalizable Unicode characters
  - characters are converted into bytes using the platform’s default character encoding or encoding given as argument to constructor

Output Stream Wrappers (4)

- `javax.crypto.CipherOutputStream`
  - Will encrypt the stream that it gets and write encrypted bytes
- `java.util.zip.ZipOutputStream`
- `java.util.zip.GZipOutputStream`
- `java.util.jar.JarOutputStream`

  Zip,Gzip, Jar output streams compresses the bytes written into them
- …

Input ASCII and binary values

- Read binary input from
  - a file, then use `FileInputStream`
  - a byte array, then use `ByteArrayInputStream`
  - a pipe written by `PipedOutputStream` in another thread, then use `PipedInputStream`
  - a String, deprecated `StringBufferInputStream`, don't use this class

Input binary

- Socket and URL input streams use a method call, rather than a constructor. Use the `getInputStream()` method.

- Reading input from
  - a socket, then use `java.net.Socket`
  - An URL, then use `java.net.URLConnection`

Input Stream Wrappers

- `java.io. DataInputStream`
- `java.io. ObjectInputStream`
- `java.io.BufferedInputStream`
  - Directly wrap the input source for best performance
- `java.io.FilterInputStream`
  - Subclass and override needed methods
- `java.io.LineNumberInputStream`
- `java.io. PushbackInputStream`
- `java.io. SequenceInputStream`
- `java.io. InputStreamReader`
- `java.util.zip.GZIPInputStream`
- …

Exercise

- Write an example that counts the number of times a particular character, such as e, appears in a file.

Formatters

- How to format the appearance
- Look at ***java.text*** package
  - Sorting order, formatting numbers and dates,
- Allows localization to Western, Arabic or Indic numbers
- Specify number of decimal places

java.text.DecimalFormat

- Constructor takes string which is the template for the format
  - a positive and negative (optional) subpattern
    - "0.00" alone is equivalent to "0.00;-0.00".
  - each subpattern has a prefix, numeric part, and suffix

```
DecimalFormat decFormat = new
   DecimalFormat("#0.00;#0.00CR");
```

- Call *format(xxx)* method

```
System.out.println(“Output = ” decFormat.format(-1.2345));
```

Output = 1.23CR

Format Strings (1)

- Symbols used in a format String
  - **`0`** a digit
  - **`#`** a digit, with zero being dropped
  - **`-`** Minus sign, indicating negative number
  - **`.`** Decimal separator or monetary decimal separator
  - **`,`** Grouping separator
  - **`;`** Separates positive and negative subpatterns
  - And others…

Format Strings (2)

- "##0.00"
  - At least one digit before decimal point, and two digits after
  - 1234.567 gives 1234.56
  - 0.123 gives 0.12

- "#.000"
  - Possible no digits before the point, three after
  - 1234.567 gives 1234.567
  - 0.123 gives .123

Format Strings (3)

- “,###”
  - Use thousand separator and no decimal places
  - 1234.567 gives 1,234
  - 0.123 gives 0

- “0.00;0.00-”
  - Show negative numbers with sign on the right
  - -1234.567 gives 1234.57-
  - 0.123 gives 0.123

Formatted Output/Input

- Java did not have any routines to do console input
- J2SE 5.0 introduces a simple text scanner to be used to read input
- J2SE 5.0 also has a new method that provides the same functionality as the printf() method in C

Formatted Output/Input: Example

- Before

```
BufferedReader br = new BufferedReader (new
  InputStreamReader (System.in));
String input = br.readLine();
int n = Integer.parseInt(input);
```

Formatted Output/Input: Scanner

- After

```
Scanner reader = new Scanner(System.in);
int n = reader.nextInt();
```

- Scanner has nextXXX() methods for reading other elements, and next() for Strings
- To process more complex input, you can use the java.util.Formatter class

Formatted Output/Input: printf

- printf() takes two arguments
  - A format String which contains fixed text you want to output
  - This String also contains format specifiers to specify how the remaining parameters should be inserted
  - Format specifiers start with a “%” and are followed characters

Formatted Output: Format Specifiers

Char	Meaning	Example	Argument	Output
s	Format as String.	%s	“string”	string
d	Format as decimal integer	%d	myInt	57
f	Format as floating point number	%f	Math.PI	3.141593
g	Format as floating point number, with scientific notation for large exponents	%g	1000000000	1.000000e+09
x	Format as hexadecimal integer	%x	(int) ‘A’	41
b	Format as boolean	%b	(i < 10)	true
c	Format as character	%c	‘A’	A
h	For debugging, print hashCode()	%h	Math.PI	144d0ce3
n	New line, use instead of \n for cross platform support of newlines	%n	no argument	\n
tX	t indicates a Date conversion. The character after the t indicates which part of the Date is	%tB	Calendar. getInstance()	March

Formatted Output: printf Examples

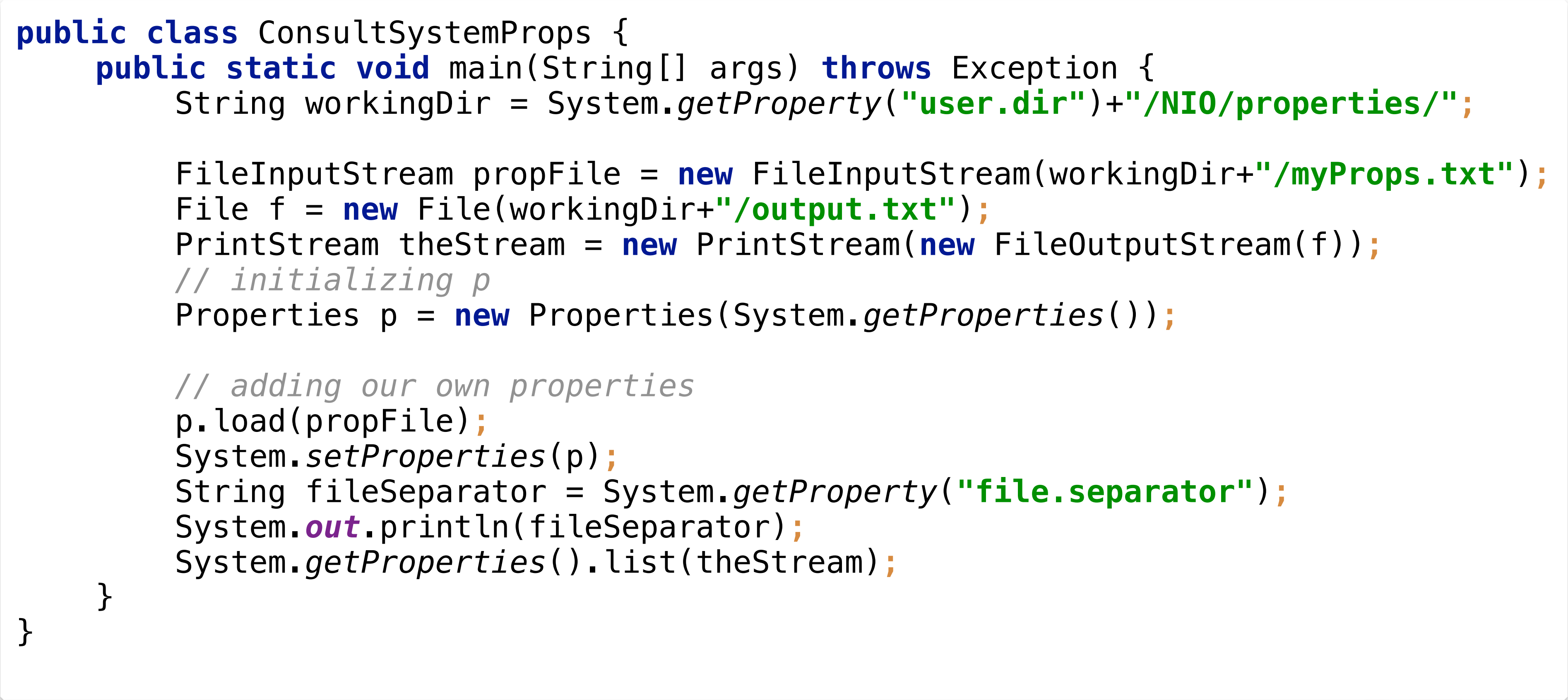
```
// printing a value
int age = 27;
System.out.printf(“Your age is: %d %n”, age);
→ Your age is: 27

// specifying minimum width and number of decimal
   places
System.out.printf(“Pi is %7.3f %n”, Math.PI);
→ Pi is       3.142

// using variable number of arguments
System.out.printf(“There are %d cows in the %s %n”,
   5, “barn”);
→ There are 5 cows in the barn
```

Properties Files

- System class maintains a set of properties
  - key/value pairs
- Two methods to read properties

```
System.getProperty(“path.separator”)
System.getProperties()
```

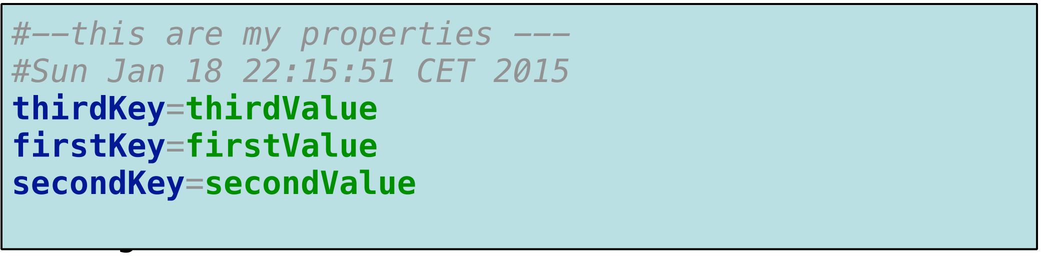
Properties Files



```java
public class ConsultSystemProps {
    public static void main(String[] args) throws Exception {
        String workingDir = System.getProperty("user.dir")+"/NIO/properties/";

        FileInputStream propFile = new FileInputStream(workingDir+"/myProps.txt");
        File f = new File(workingDir+"/output.txt");
        PrintStream theStream = new PrintStream(new FileOutputStream(f));
        // initializing p
        Properties p = new Properties(System.getProperties());

        // adding our own properties
        p.load(propFile);
        System.setProperties(p);
        String fileSeparator = System.getProperty("file.separator");
        System.out.println(fileSeparator);
        System.getProperties().list(theStream);
    }
}
```

Properties for a user program

```
public class PropertiesWriter {


    public static void setProperties() {
        String workingDir = System.getProperty("user.dir")+"/NIO/properties/";
        try {
            Properties applicationProps = new Properties();
            FileOutputStream out = new FileOutputStream(workingDir+
                    "/application.properties");
            applicationProps.put("firstKey", "firstValue");
            applicationProps.put("secondKey", "secondValue");
            applicationProps.put("thirdKey", "thirdValue");
            applicationProps.store(out, "--these are my properties ---");
            out.close();
        } catch (IOException ioe) {
            System.out.println("An IO exception occurred: " + ioe.getMessage());
        }
        finally{

        }
    }

    public static void main(String
        setProperties();
    }
}
```



```
#--this are my properties ---
#Sun Jan 18 22:15:51 CET 2015
thirdKey=thirdValue
firstKey=firstValue
secondKey=secondValue
```

Properties for a user program

```
public class PropertiesReader {
    public static void readProperties(String propFile) {
        try {
            Properties theProps = new Properties();
            FileInputStream in = new FileInputStream(propFile);
            theProps.load(in);
            System.out.println("The property with the first key is "
```

```
The property with the first key is firstValue
Is the property with the second key present? : true
Here comes the whole list :
-- listing properties --
thirdKey=thirdValue
firstKey=firstValue
secondKey=secondValue
All elements
thirdValue
firstValue
secondValue
There are 3 properties
```

```
    }

    public static void main(String[] args) {
        String workingDir = System.getProperty("user.dir")+"/NIO/properties/";
        readProperties(workingDir+"/application.properties");
    }
}
```

Questions ??