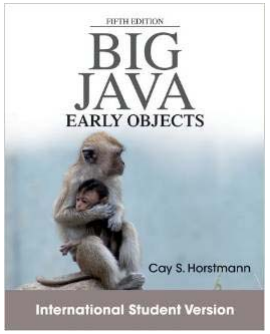
Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Module BTI7055: Object-Oriented Programming 2

J.-P. Dubois & S. Kramer
BFH-TI: Computer Science Division / 2017-2018

Input / Output

jean-paul.dubois@bfh.ch



Based on *Big Java, 5th Edition* by Cay Horstmann

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object Input and Output Streams

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object input and Output Streams

Reading and Writing Text Files

- To read files that contain text, use the `java.io.File` and the `java.util.Scanner` classes:
  - First, construct a `File` object with the name of the input file
  - Then, use the `File` object to construct a `Scanner` object
  - Finally, use the `Scanner` methods (e.g. `next`, `nexLine`, `nextInt` or `nextDouble`) to read the data from the input file

- Example: Process numbers from an input file

```
File inputFile = new File("input.txt");
Scanner in = new Scanner(inputFile);
while (in.hasNextDouble()) {
  double value = in.nextDouble();
  ... Process value ...
}
```

Reading and Writing Text Files

- To write output to a file, use the `PrintWriter` class:
  - Construct a `PrintWriter` object with the name of the output file
  - If the file already exists, it is emptied(!) before the new data are written into it.
  - If the file does not exist, an empty file is created.

- `PrintWriter` is an enhancement of `PrintStream`. Hence, you can use the `print`, `println` and `printf` methods with any `PrintWriter` object.

```
PrintWriter out = new PrintWriter("output.txt");
out.println("Hello, World!");
out.printf("Total: %8.2f\n", total);
```

Reading and Writing Text Files

- When you are done processing a file, be sure to close the Scanner or PrintWriter. That will close the associated files.

```
in.close();
out.close();
```

- However, the easiest way to make sure that the files will be closed when the processing is over is to use the try-with-resource statement.

```
try (Scanner in = new Scanner(new File("input.text");
     PrintWriter out = new PrintWriter("output.txt")) {
  // ... Process the files ...
}  catch (FileNotFoundException ex) { ... }
```

Reading and Writing Text Files

- When the `try` block ends, the `close` method is automatically invoked on the resource(s), whether or not an exception has occured. Note that:
  - In a try-with-resources statement, you may declare more than one resource, separated by semicolons. When the `try` block ends, they will be closed in the opposite order of their creation.
  - The resource(s) declared in the try-with-resources statement must implement the interface `java.lang.AutoCloseable`.
  - A try-with-resources statement can have `catch` and `finally` clauses just like an ordinary `try` statement. Any `catch` or `finally` block is run after the resources declared have been closed.

Reading and Writing Text Files

- ⚠ Beware:
  - If you terminate the program execution without closing the PrintWriter, not all of the output may be written to the disk file.
  - Should you want to force the output to be written without closing the file, call:

```
out.flush();
```

- Example:
  - The following program reads a file containing numbers and writes the numbers, lined up in a column and followed by their total to another file.

Reading and Writing Text Files

- Input file content:

```
32 54 67.5 29 35 80
115 44.5 100 65
```

- Output file:

```
            32.00
            54.00
            67.50
            29.00
            35.00
            80.00
           115.00
            44.50
           100.00
            65.00
 Total:    622.00
```

Total.java

```
import java.io.*;
import java.util.*;

/**
   This program reads a file with numbers, and writes the numbers to another
   file, lined up in a column and followed by their total.
*/
public class Total {

  public static void main(String[] args) {
    // Prompt for the input and output file names
    Scanner console = new Scanner(System.in);
    System.out.print("Input file: ");
    String inputFileName = console.next();
    System.out.print("Output file: ");
    String outputFileName = console.next();
```

Total.java *(cont.)*

```
    // Construct the Scanner and PrintWriter objects for reading and writing
    File inputFile = new File(inputFileName);
    try (Scanner in = new Scanner(inputFile);
         PrintWriter out = new PrintWriter(outputFileName)) {
      // Read the input and write the output
      double total = 0;

      while (in.hasNextDouble()) {
        double value = in.nextDouble();
        out.printf("%15.2f\n", value);
        total += value;
      }
      out.printf("Total: %8.2f\n", total);

    } catch (FileNotFoundException ex) {
      // Handle the exception
      System.out.println(ex.getMessage());
    }
  }
}
```

Reading and Writing Text Files

- Reading Web Pages:
  - You can read the content of a web page as follows:

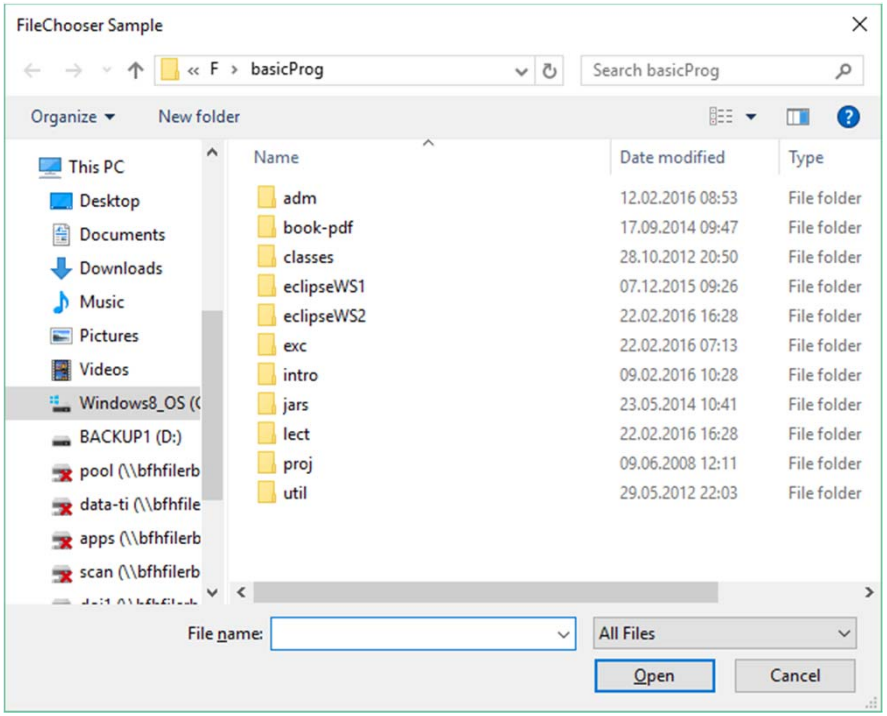
```
String address = "http://horstmann.com/index.html";
URL pageLocation = new URL(address);
Scanner in = new Scanner(pageLocation.openStream());
... Read the web page content with the Scanner ...
```

  - You will have to import the `java.net` package and handle the `IOException` that may thrown by the `openStream` method.

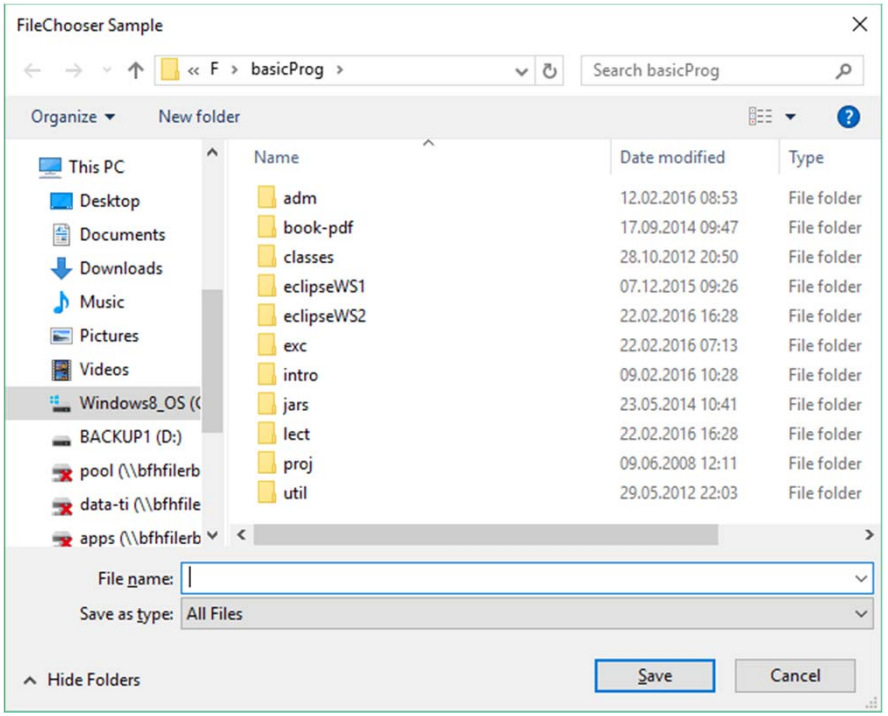
Reading and Writing Text Files

- In a program with a GUI, you can use the JavaFX `FileChooser` dialog (in package `javafx.stage`) to select files:

Opened with `showOpenDialog`



Opened with `showSaveDialog`



Reading and Writing Text Files

- To construct a `FileChooser`, call the `showOpenDialog`, `showOpenMultipleDialog` or `showSaveDialog` method, depending on your need (passing the owner window to them).
  - These methods return the selected file(s), or `null` if no file has been selected (see API).
  - You can define extension filters, to filter which files can be selected, based on the file name extensions.

```
FileChooser chooser = new FileChooser();
chooser.setInitialDirectory(new File("\\F\\basicProg"));
chooser.getExtensionFilters()
    .addAll(new FileChooser.ExtensionFilter("All Files", "*.*"),
            new FileChooser.ExtensionFilter("Java Files", "*.java"));
File selectedFile = chooser.showOpenDialog(stage);
if (selectedFile != null) {  ...  }
Platform.exit(); // Force exit
```

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object input and Output Streams

Text Input and Output

- Reading Words
  - The `Scanner.next` method reads a word at a time. For example, consider the loop:

```
while (in.hasNext()) {
  String input = in.next();
  System.out.println(input);
}
```

  - Applied on the input:

```
Hello, world! How are you?
```

  - … the output will be:

```
Hello,
world!
How
are
you?
```

Text Input and Output

- A *word* is any sequence of characters delimited by whitespace.
- *Whitespace* includes spaces, tab characters and the newline character.
- When the next method is executed, the whitespace characters that precede the word are *consumed* (i.e. removed from input), but they do not become part of the word.
- The first character that is not whitespace becomes the first character of the word. More characters are added until either another white space character occurs, or the end of the input has been reached.

Text Input and Output

- The scanner can also use delimiters other than whitespace. You can define delimiters by calling the `useDelimiter` method.
  - For example: to concatenate the sentence:

    `Va, cours, vole, et nous venge!`

    as follows:

    `_Va_cours_vole_et_nous_venge!`

    You could use the following code:

```
Scanner in = new Scanner(...);
// New delimiter: commas + 1 whitespace OR 1 whitespace
in.useDelimiter(", | ");
while (in.hasNext())
  System.out.print("_" + in.next())
```

Text Input and Output

- Reading Characters:
  - As another example, `useDelimiter` gives you a simple way to read all characters of a file.

```
Scanner in = new Scanner(...);
in.useDelimiter("");
while (in.hasNext()) {
   char ch = in.next().charAt(0);
   ... Process value ...
}
```

Text Input and Output

- Classifying Characters:
  - The `Character` class offers several useful `static` methods to find out of what kind a character is. For example, `Character.isDigit(ch)` returns `true`, if `ch` is a digit.

Method	Accepted Characters (examples)
`isDigit`	0, 1, 2
`isLetter`	A, B, C, a, b, c
`isUpperCase`	A, B, C
`isLowerCase`	a, b, c
`isWhiteSpace`	space, newline, tab

Text Input and Output

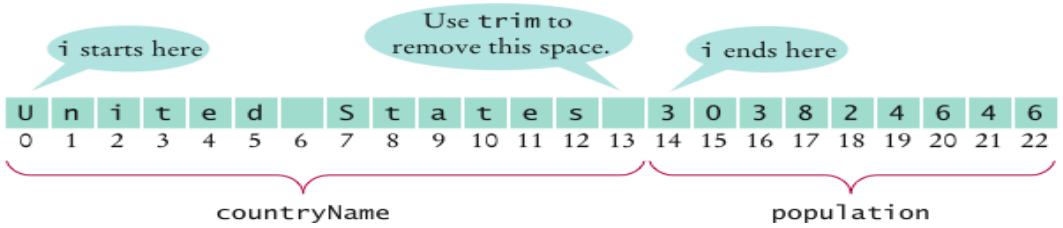
- Reading Lines
  - The nextLine method reads a line of input and consumes the newline character at the end of the line.
  - Example: process a file with population data with lines like this:

```
China  1330044605
India  1147995898
United States 303824646
...
```

  - For each data line, you can proceed as follows:

```
// 1. Read the entire line into a string
String line = in.nextLine();
```

Text Input and Output




```
// 2. Find out where the name ends and the number starts.
int i = 0;  while (!Character.isDigit(line.charAt(i))) { i++ };

// 3. Extract the country name and population.
String countryName = line.substring(0, i);
String population = line.substring(i);

// 4. Remove spaces at end of the country name and pop.
countryName = countryName.trim();
population = population.trim();

// 5. Convert the population string to a number.
int populationValue = Integer.parseInt(population);
```

Text Input and Output

- Note that the following approach could be easier.

```
// 1. Read the entire input line into a string.
String line = in.nextLine();

// 2. Construct a Scanner object to read the string.
Scanner lineScanner = new Scanner(line);

// 3. Extract country name and population
String countryName = lineScanner.next();

// 4. We must also read the country names which
// consist of more than one word
while (!lineScanner.hasNextInt())
  countryName = countryName + " " + lineScanner.next();
int populationValue = lineScanner.nextInt();
```

Text Input and Output

- Reading Numbers:
  - To read numbers, use the `nextInt`, resp. `nextDouble`, methods. They consume white space and the next word, which must be a properly formatted (`int`, resp. `double`) number:

```
double value = in.nextDouble();
```

  - If the word read is not a number, an "input mismatch exception" occurs.
  - Of course, you can detect if the next item to read is a number by using the predicate methods `hasNextInt` and `hasNextDouble`.
  - Note that the `nextInt`, `nextDouble`, and `next` methods do not consume the white space that follows the number or word.

Text Input and Output

- Example – Suppose a file with country names and population values in this format:

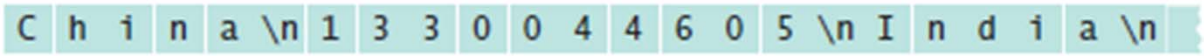
```
China
1330044605
India
1147995898
...
```

- Use the following loop:

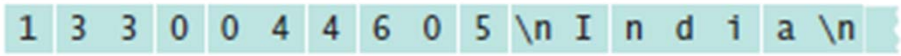
```
while (in.hasNextLine()) {
  String countryName = in.nextLine();
  int population = in.nextInt();
  in.nextLine(); // Consume the remaining newline
  ... Process the country name and population ...
}
```

Text Input and Output

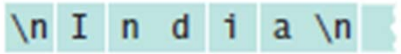
- Initial input:



- Input after first call to `nextLine`:



- Input after call to `nextInt`:



- → Before reading "India", it is necessary to consume the remaining newline on the left side.

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object input and Output Streams

Command Line Arguments

- Command line arguments:
  - When you invoke a Java program from the command line, you can enter some additional *command line arguments.*

```
java ProgramClass -v input.dat
```

  - These arguments are placed one to one into the **`args`** array of the **`main`** method. They can then be processed as normal strings by the **`main`** method.

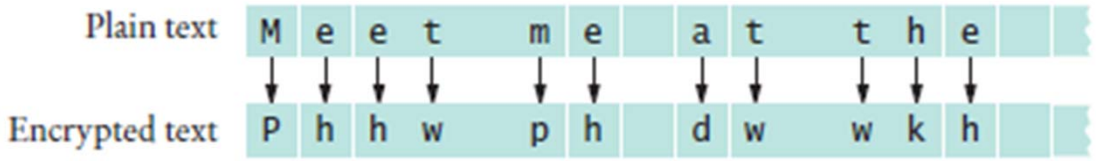
`args[0]` ← `"-v"` `args[1]` ← `"input.dat"`

  - If more than one parameter must be assigned to the same **`args`** element, enclose them in quote marks " … ". Also use quote marks, if a parameter has a special meaning for the command line interpreter:

```
java Calculator "Multiplication example" 25 "*" 38
```

Command Line Arguments

- Example: A program that encrypts a file, using a *Caesar cipher* that replaces A with a D, B with an E, and so on.



  - The program will take the following command line arguments:
    - An optional -d flag to indicate decryption instead of encryption
    - The input file name
    - The output file name
  - For example:
    - Encrypt the file `input.txt` and place the result into `encrypt.txt`

```
java CaesarCipher input.txt encrypt.txt
```

    - Decrypt the file `encrypt.txt` and place the result into `output.txt`

```
java CaesarCipher -d encrypt.txt output.txt
```

CaesarCipher.java

```
import java.io.*;
import java.util.*;
/**
   This program encrypts a file using the Caesar cipher.
*/
public class CaesarCipher {

  /**
     Prints a message describing proper usage.
  */
  private static void usage() {
    System.out.println("Usage: java CaesarCipher [-d] <infile> <outfile>");
    System.exit(0); // Terminate execution
  }

  public static void main(String[] args) throws FileNotFoundException {
    final int DEFAULT_KEY = 3;
    int key = DEFAULT_KEY;
    int argNum = 0; // Number of the current command line argument
```

CaesarCipher.java *(cont.)*

```
   if (argNum < args.length && args[argNum].charAt(0) == '-') {
     // This a command line option
     if (args[argNum].length() == 2 && args[argNum].charAt(1) == 'd') key = -key;
     else usage();
     argNum++;
   }

   // Read the file names
   if (args.length != argNum + 2) usage();
   String inFile = args[argNum++];
   String outFile = args[argNum];
   try (Scanner in = new Scanner(new File(inFile));
     PrintWriter out = new PrintWriter(outFile)) {
     in.useDelimiter(""); // Process individual characters
     while (in.hasNext()) {
       char from = in.next().charAt(0);
       char to = encrypt(from, key);
       out.print(to);
     }
   }
}
```

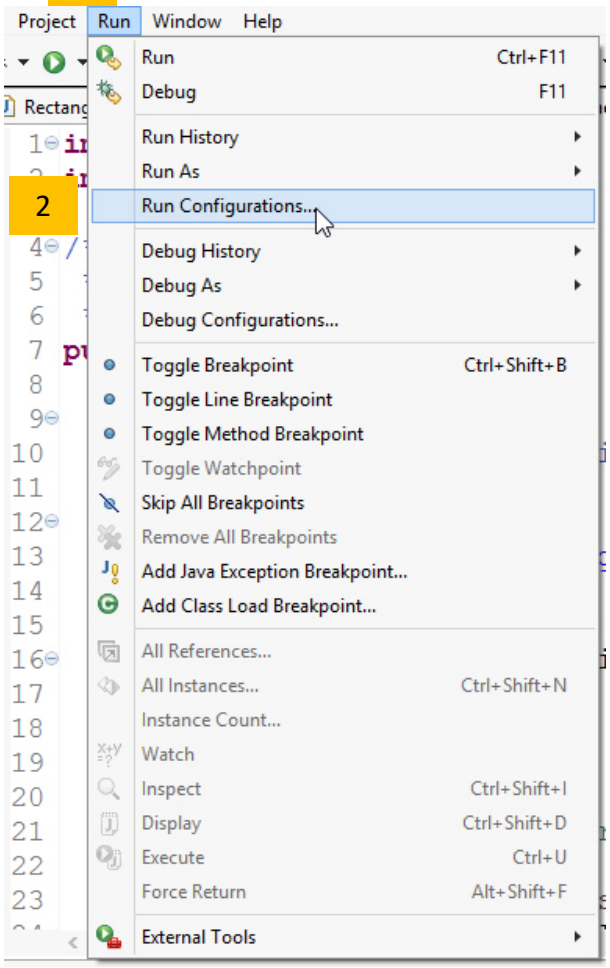
CaesarCipher.java *(cont.)*

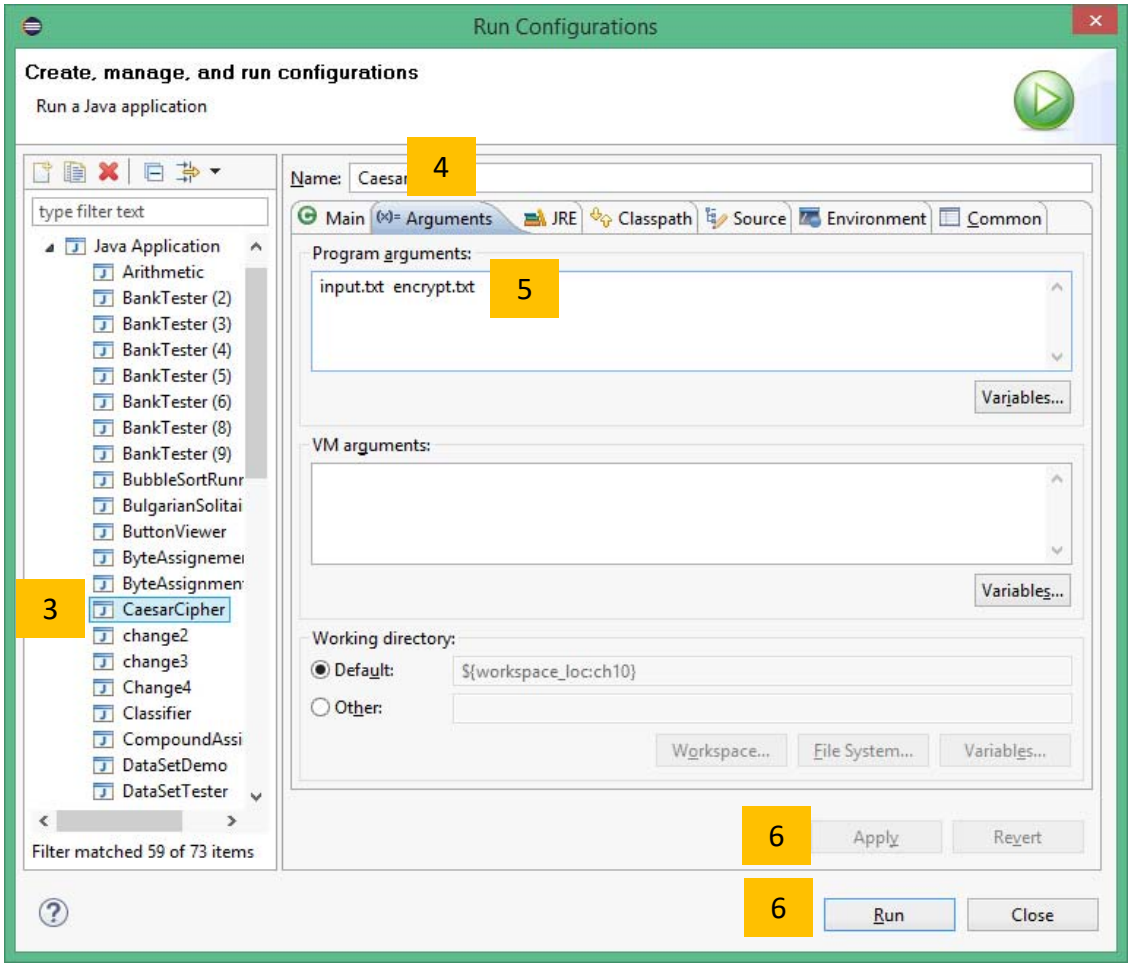
```
   /**
      Encrypts upper- and lowercase characters by shifting them
      according to a key.
      @param ch the letter to be encrypted
      @param key the encryption key
      @return the encrypted letter
   */
   private static char encrypt(char ch, int key) {
     int base = 0;
     if ('A' <= ch && ch <= 'Z') base = 'A';
     else if ('a' <= ch && ch <= 'z') base = 'a';
     else return ch; // Not a letter

     int offset = ch - base + key;
     final int LETTERS = 26; // Number of letters in the Roman alphabet
     if (offset >= LETTERS)
       offset -= LETTERS;
     else if (offset < 0)
       offset += LETTERS;
     return (char) (base + offset);
   }
}
```

Command Line Arguments

Eclipse: Command line arguments





Command Line Arguments

- Command line arguments with JavaFX:
  - To read command line arguments in a JavaFX application, you can use the `Application.Parameters` class:

```
public class MyFXApplication extends Application {

  public void start(Stage stage) {
    Parameters params = getParameters();
    List<String> paramList = params.getRaw();
    ...
  }
}
```

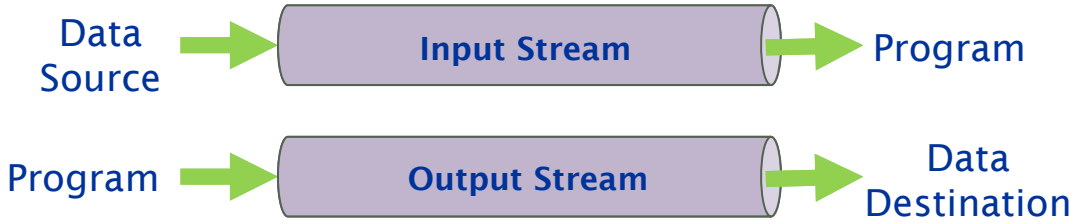
  - The arguments are stored in `paramList` in a similar way as before in the `args` array. The list can be empty, but is never `null`.

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object input and Output Streams

I/O Streams

- Most applications need to process some input and produce some output based on that input. The purpose of the classes in the `java.io` package is to make that possible in Java.
- To execute IO operations, Java uses the concept of IO *streams*, which are flows of data that can either be read from or written to.
- Streams are typically connected to a data source or data destination, like a file, network connection, memory buffer, etc.
- A stream is a continuous flow of data that can be read or written sequentially. You cannot move back and forth in a stream.



I/O Streams

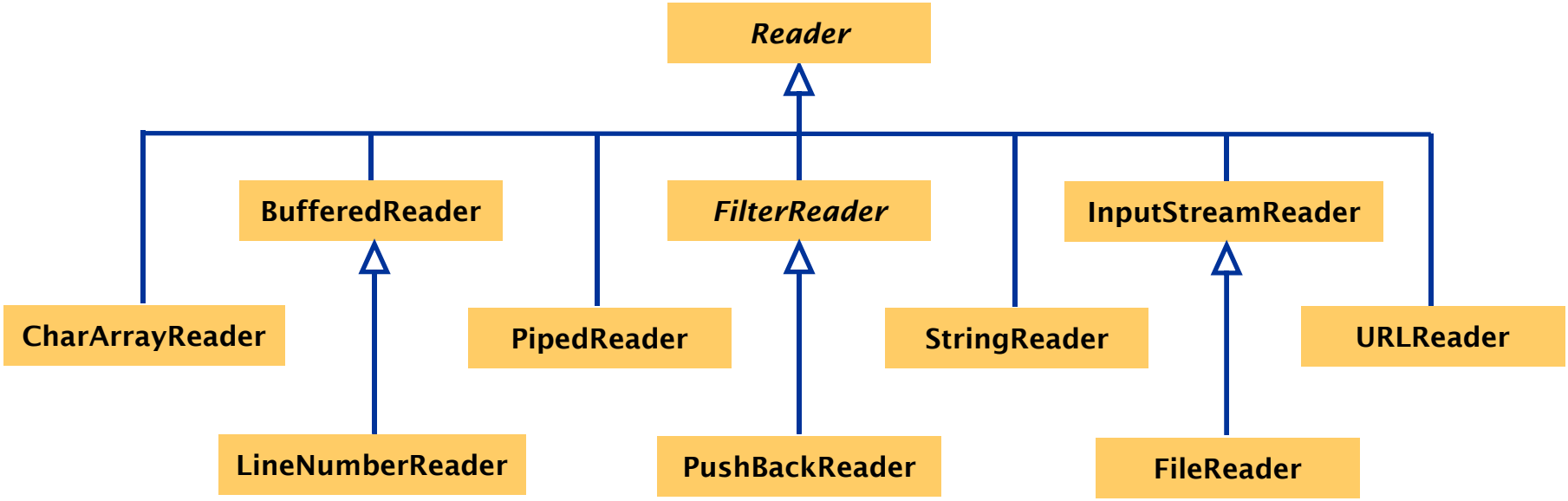
- There are two fondamentally different ways to store data:
  - In text format
  - In binary format
- In *text form, the* data items are represented in human readable form, as a sequence of *characters*
  - Example: 12345 is stored '1' '2' '3' '4' '5' (five characters)
- In *binary form, the* data items are represented in *bytes*
  - Example: 12345 (**`int`**) is stored as 0 0 48 57 (four bytes)
- These two kind of data formats are handled in Java by two different groups of classes:
  - The *character stream* classes (for the text format)
  - The *byte stream* classes (for the binary format)

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object input and Output Streams

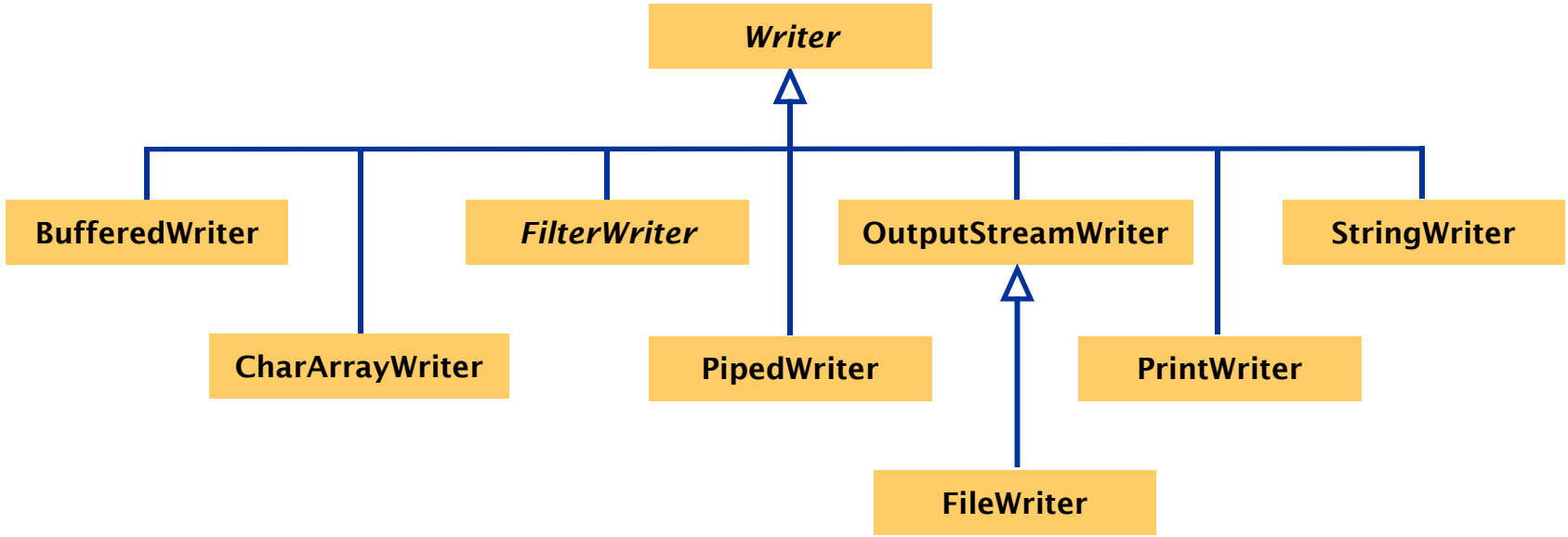
Readers and Writers

- Text format
  - To process input and output of information stored in text form, the Java library provides the `Reader` and `Writer` class sets.
- The *Reader* Character Streams



Readers and Writers

- The *Writer* Character Streams



Readers and Writers

- For reading text, however, it is in most cases more convenient to use the `Scanner` class instead of `Reader` classes.
  - Internally, the `Scanner` class makes use of readers to read characters.
  - You can construct a `Scanner` object for any object of type `File`, `InputStream`, `Reader` (more exactly: `Readable`) or `String`.
- For writing text and numbers in text format, the most convenient class is the `PrintWriter` class.
  - You can construct a `PrintWriter` object for any object of type `File`, `OutputStream`, `Writer` or `String`.

Readers and Writers

- When manipulating text strings, you need to consider the *character encoding.*

- Example of Unicode character encoding
  - In the table below, *UTF-8* and *UTF-16* are two different encoding schemata for Unicode characters

Character	UTF-8: hex (dec)	UTF-16: hex (dec)
e	65 (101)	00 65 (0 101)
é	C3 A9 (195 169)	00 E9 (0 233)

- The `Reader` and `Writer` hierarchies have one class each that is responsible for converting between bytes and characters.

- By default, these classes use the default character encoding of the computer executing the program.

Readers and Writers

- The following classes perform the conversions between bytes and characters.
  - The `OutputStreamWriter` class turns a stream of Unicode characters into a stream of bytes, using a chosen encoding.
  - Conversely, the `InputStreamReader` class turns a byte stream that contains bytes specifying characters in some encoding into a Unicode character stream.
- To achieve the input or output operations, the appropriate `Reader` or `Writer` streams must be chained together.
- However, in most cases, this chaining can remain transparent to the programmer if approriate classes or constructors are used.

Readers and Writers

- Example 1: To store data into a text file, you had to chain the following output character streams together:

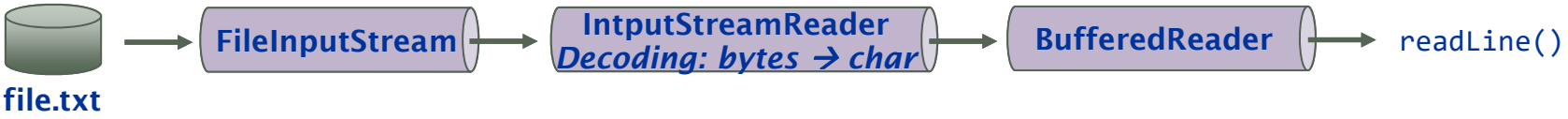

```
PrintWriter pw = new PrintWriter(new OutputStreamWriter(new FileOutputStream("file.txt")));
```

- By selecting a convenient PrintWriter constructor, you can store the data as follows (e.g. by using the UTF-16 encoding):

```
String[] text = {.....};
// Write text lines into "file.txt" with UTF-16 encoding
PrintWriter pw = new PrintWriter("file.txt", "UTF-16"));
for (String s : text)
  pw.println(s);
```

Readers and Writers

- Example 2: To read lines of text from a file in a program, you had to chain the following input character streams together:




```
BufferedReader br = new BufferedReader(new InputStreamReader(new FileInputStream("file.txt")));
```

- To read a text stored in a file, use preferably a Scanner as follows (e.g. to read data encoded in UTF-16):

```
// Read "file.txt" using UTF-16 encoding
Scanner in = new Scanner(new File("file.txt"), "UTF-16");
while(in.hasNext()) {
  String s = in.next();
  ... Process the data ...
}
```

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object input and Output Streams

Binary Input and Output

- Binary format:
  - The binary format allows a more compact representation and a more efficient handling of the data as the text format.
  - To process input and output use the *byte streams* supported by the `InputStream` and `OutputStream` classes and their subclasses.
  - The `InputStream`, resp. `OutputStream`, class has a method, `read`, resp. `write`, to process a single byte at a time

Binary Input and Output

- Example 1 - Read data from a disk file as a sequence of bytes
  - By each call, the `read` method returns the next byte from the file. At the end of file, `-1` is returned

```
InputStream in = new FileInputStream("input.dat");
int next = in.read();
while (next != -1) {
   ... Process "next" ...
   next = in.read();
}
```

  - Note that you can read some number of bytes from an `InputStream` and store them into a buffer as follows:

```
byte[] buffer = new byte[...];
int numberOfBytesRead = in.read(buffer); //Try to fill up the buffer
```

Binary Input and Output

- Example 2 - Write data to a disk file as a sequence of bytes

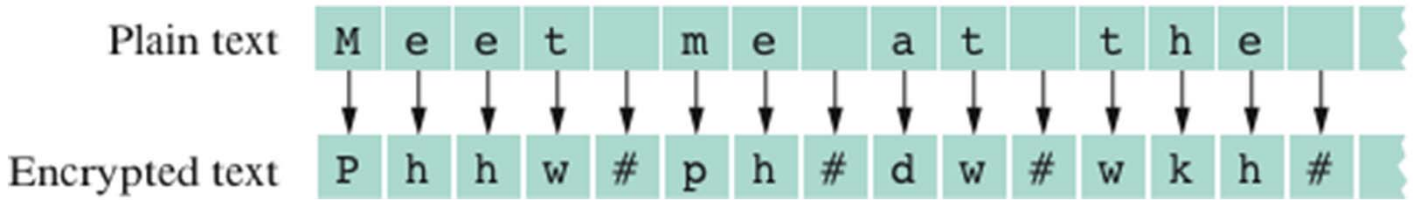
```
OutputStream out = new FileOutputStream("output.dat");
while (more data to process) {
  byte b = ...;
  out.write(b);
}
```

  - Note that you can write an array of bytes to an `OutputStream` as follows:

```
byte[] buffer = new byte[...];
... Fill the buffer ...
out.write(buffer);
```

Binary Input and Output

- A complete example: An encryption program
  - The program scrambles the bytes in a file so that it is readable only to those who know the encryption method and secret keyword
  - We use again the Caesar cipher, but now we encode all bytes, making the file unreadable by a text editor
  - We choose *encryption key*, a number between 1 and 25, that indicates the shift to be used in encrypting each byte



  - To decrypt, use the negative of the encryption key.

CaesarCipher.java

```
import java.io.*;

/**
   This class encrypts files using the Caesar cipher.
   For decryption, use an encryptor whose key is the
   negative of the encryption key.
*/
public class CaesarCipher {

  private int key;

  /**
     Constructs a cipher object with a given key.
     @param aKey the encryption key
  */
  public CaesarCipher(int aKey) {
    key = aKey;
  }
```

CaesarCipher.java *(cont.)*

```
/**
   Encrypts the contents of a stream.
   @param in the input stream
   @param out the output stream
*/
public void encryptStream(InputStream in, OutputStream out)
                                                    throws IOException {
  boolean done = false;
  while (!done) {
    int next = in.read();
    if (next == -1)
      done = true;
    else {
      int encrypted = encrypt(next);
       out.write(encrypted);
    }
  }
}
```

CaesarCipher.java *(cont.)*

```
 /**
    Encrypts a value.
    @param b the value to encrypt (between 0 and 255)
    @return the encrypted value
  */
  public int encrypt(int b) {
    return (b + key) % 256;
  }
}
```

CaesarEncryptor.java

```
import java.io.*;
import java.util.*;

/**
   This program encrypts a file, using the Caesar cipher.
*/
public class CaesarEncryptor {

  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.print("Input file: ");
    String inFile = in.next();
    System.out.print("Output file: ");
    String outFile = in.next();
    System.out.print("Encryption key: ");
    int key = in.nextInt();
```

CaesarEncryptor.java *(cont.)*

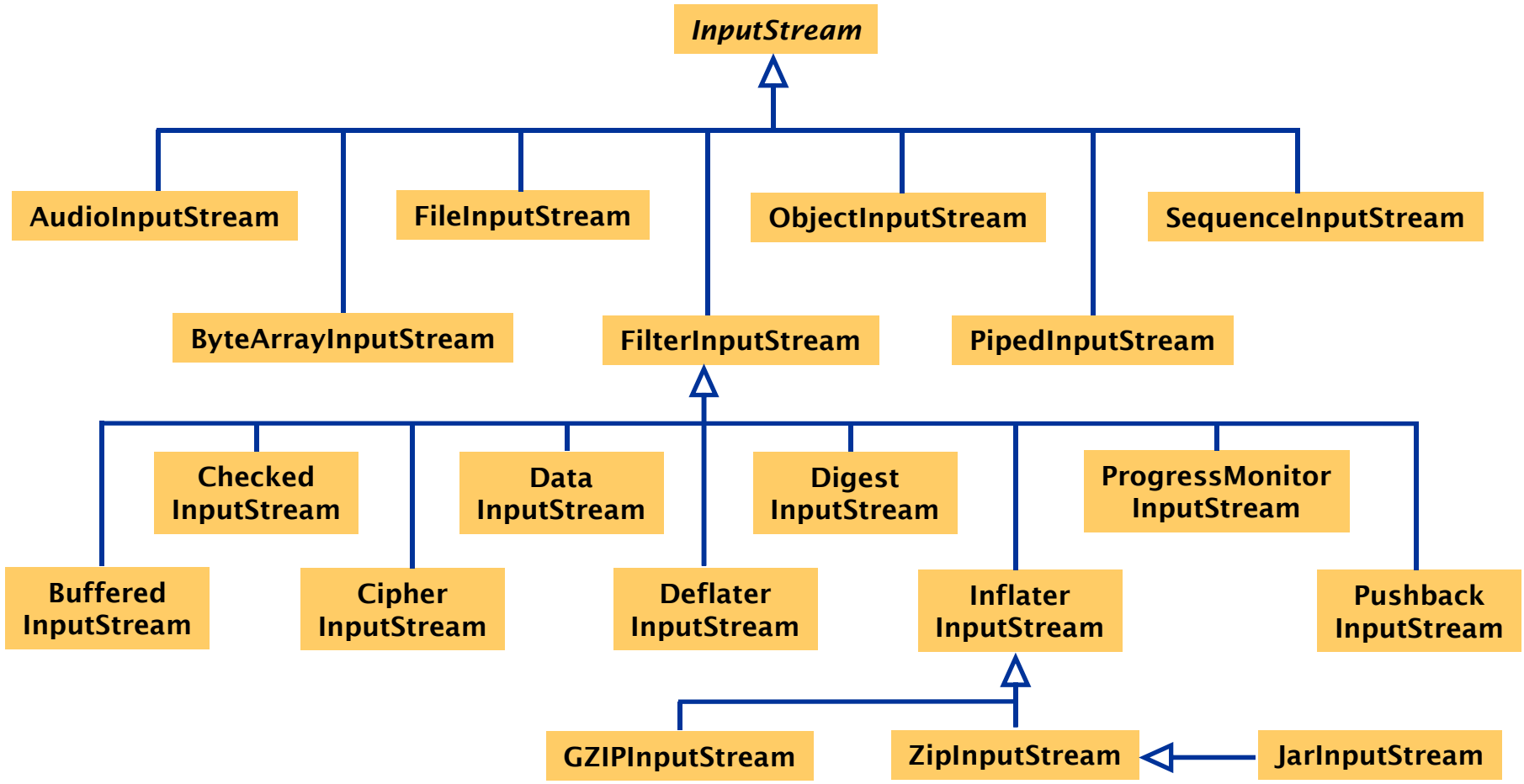
```
 try (InputStream inStream = new FileInputStream(inFile);
      OutputStream outStream = new FileOutputStream(outFile)) {

      CaesarCipher cipher = new CaesarCipher(key);
      cipher.encryptStream(inStream, outStream);

    } catch (IOException ex) {
      System.out.println("Error processing file: " + ex);
    }
  }
}
```

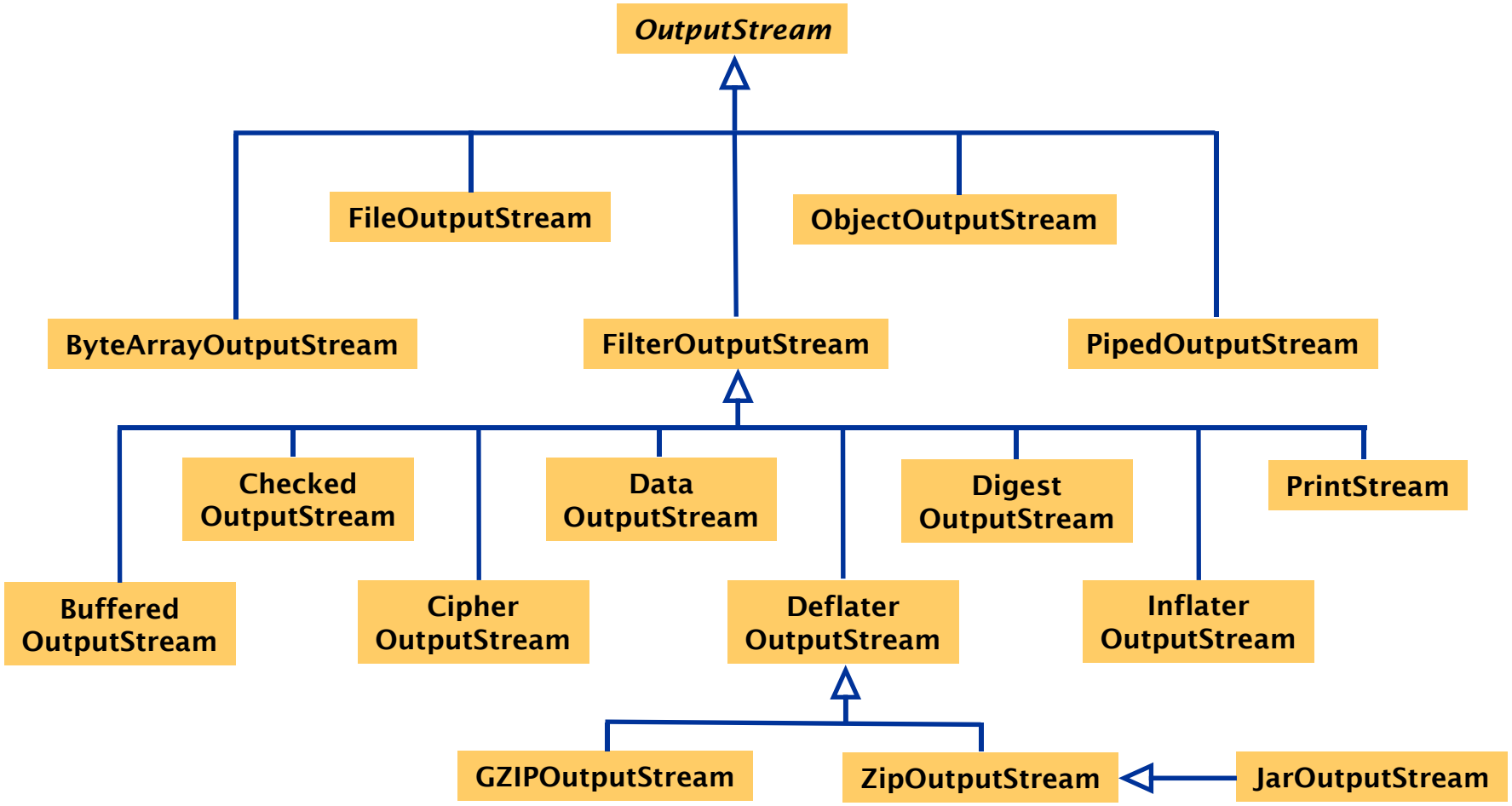
Binary Input and Output

- The *InputStream* byte Streams



Binary Input and Output

- The *OutputStream* byte Streams



Binary Input and Output

- Read the Java API to find out the exact functionality of the input and output streams. Some examples:
  - The `ByteArrayInputStream` allows you to read data from byte arrays as streams; the `ByteArrayOutputStream` allows you to capture data written to a stream in an array.
  - The `BufferedInputStream` and `BufferedOutputStream` provide buffering to your byte streams. Buffering can speed up IO quite a bit, especially for disk access and larger data amounts. You can specify the size of the buffer in the constructor.

```
InputStream bis = new BufferedInputStream(
    new FileInputStream("input.dat"), 8 * 1024);// Buffer: 8 kB

OutputStream bos = new BufferedOutputStream(
    new FileOutputStream("output.dat"));// Buffer: Default size
```

Binary Input and Output

- The `DataOutputStream` contains methods for writing, numbers, characters, `boolean` values or strings in binary format. The `DataInputStream` contains the corresponding read methods (except for strings).

```
DataOutputStream dos = new DataOutputStream(
                                    new FileOutputStream("file.dat");
dos.writeInt(12345);
dos.writeDouble(123.45);
String s = "Hello!";
dos.writeByte(s.length()); // Store the string length
dos.writeChars(s); // ... before the string
...
DataInputStream dis = new DataIntputStream(
                                    new FileInputStream("file.dat");
int i = dis.readInt();
double d = dis.readDouble();
byte b = in.readByte(); // Get the string length
s = "";
for (int j = 0; j < b; j++) s += dis.readChar(); //...to read the string
```

Binary Input and Output

- The `PushbackInputStream` adds the ability to another input stream to "push back" or "unread" one byte, so that the next read operation on the input stream will reread the byte that was pushed back.

- The `SequenceInputStream`, concatenates two or more other input streams. First, all bytes from the first input stream are read, then the bytes of the second one, etc.

```
InputStream input1 = new FileInputStream("file1.dat");
InputStream input2 = new FileInputStream("file2.dat");
SequenceInputStream sis = new SequenceInputStream(input1, input2);
```

Binary Input and Output

- As by the character streams, you can chain the appropriate streams together to achieve the input or output operations you need:
  - Example: Output stream to store buffered int values into a file:


```
DataOutputStream dos = new DataOutputStream(
                           new BufferedOutputStream(new FileOutputStream("file.dat")));
dos.writeInt(...);
```

  - Input stream to read these values again:


```
DataInputStream dis = new DataInputStream(
                          new BufferedInputStream(new FileInputStream("file.dat")));
int ... = dis.readInt(...);
```

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object input and Output Streams

File and Directory Operations

- To execute file or directory operations, you can use the `File` class. However, the `Path` interface and the `Files` and `Paths` classes (in package `java.nio.file`) offer more operations and overcome some limitations of the `File` class.

- Paths
  - A `Path` describes the location of a file or directory:

```
Path sourcePath = Paths.get("IODemo.java");
Path dirPath = Paths.get("/basicProg/oop2/examples");
```

  - A `Path` can also be specified as a sequence of directory names, optionally followed by a file name:

```
Path dirPath = Paths.get("/basicProg", "oop2", "examples");
```

File and Directory Operations

- Note that a `Path` instance does not have to correspond to an existing file. It is merely an abstract sequence of names.

- You combine paths with the `resolve` method. The argument to resolve can be a `Path` or a string:

```
// fullPath is: /basicProg/oop2/examples/IODemo.java
Path fullPath = dirPath.resolve(sourcePath);
```

- With `toAbsolutePath`, you can turn a relative directory into an absolute one:

```
// absolutePath is: c:/basicProg/oop2/examples/IODemo.java
Path absolutePath = Paths.get("IODemo.java").toAbsolutePath();
```

File and Directory Operations

- There are also methods for taking paths apart:

```
// parent is the path: /basicProg/oop2/examples
Path parent = fullPath.getParent();
// filename is the path: IODemo.java
Path filename = fullPath.getFileName();
```

- As Path extends the `Iterable<Path>` interface, you can also use the enhanced for loop with a `Path`:

```
for (Path p : fullPath) {
   ...p is set to basicProg, oop2, examples and IODemo.java ...
 }
```

File and Directory Operations

- Creating and Deleting Files and Directories
  - Create a directory or a file (path is a `Path` object).
    - Note that if you try to create a file or directory that already exists, an exception is thrown:

```
// Create a new directory
// (all but the last component in the path must exist)
Files.createDirectory(path);
// Create a directory path
Files.createDirectories(path);
// Create an empty file
Files.createFile(path);
```

  - Test if a file or a directory exists:

```
boolean exists = Files.exists(path);
```

File and Directory Operations

- Find out whether an existing path is a file or a directory:

```
boolean isDirectory = Files.isDirectory(path);
boolean isFile = Files.isRegularFile(path);
```

- Delete a file or an empty directory:

```
Files.delete(path);
```

- If the file could possibly not exist, use:

```
boolean deleted = Files.deleteIfExists(path);
```

File and Directory Operations

- Useful File Operations
  - Yield the size of a file in bytes:

```
long size = Files.size(path)
```

  - Read an entire file into a list of lines (text file) or a byte array (binary file):

```
List<String> lines = Files.readAllLines(path);
byte[] bytes = Files.readAllBytes(path);
```

  - Write / Append a collection of lines or an array of bytes to a file:

```
Files.write(path, lines); // Write to a new file
Files.write(path, bytes);
Files.write(path, lines, StandardCopyOption.APPEND); // Append
Files.write(path, bytes, StandardCopyOption.APPEND);
```

File and Directory Operations

- Read a file into a single string. If necessary, you can give the appropriate file encoding (e.g. UTF-8):

```
String contents = new String(Files.readAllBytes(path), "UTF-8");
```

- Save a string to a (e.g. UTF-8) file:

```
Files.write(path, contents.getBytes("UTF-8"));
```

- Get a byte or a character stream:

```
InputStream inputStream = Files.newInputStream(path);
OutputStream outputStream = Files.newOutputStream(path);
Reader reader = Files.newBufferedReader(path);
Writer writer = Files.newBufferedWriter(path);
```

File and Directory Operations

- Copy or move a file (to move a non-empty directory, you have to move all ist descendants):

```
Files.copy(fromPath, toPath);
Files.copy(inputStream, toPath);
Files.copy(fromPath, outputStream);
Files.move(fromPath, toPath);

// You can also specify various options. For example,
// to overwrite an existing target by copying use:
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING);
```

File and Directory Operations

- Example:
  - Store a Web page to a file

```
URL url = new URL("http://horstmann.com/index.html");
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
Path path = Paths.get("Horstmann.html");
Files.copy(in, path);
```

File and Directory Operations

- Visiting Directories
  - To get a list of the files and directories contained in a directory, proceed as follows (`Stream` and `Collectors` are in the package `java.util.stream`):

```
try (Stream<Path> entries = Files.list(dirPath)) {
  List<Path> paths = entries.collect(Collectors.toList());
  ... Process the list paths ...
}
```

  - `Files.list` does not visit subdirectories. In order to get all descendant files and directories, call `Files.walk`.

Outline

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- I/O Streams
- Readers and Writers
- Binary Input and Output
- File and Directory Operations
- Object input and Output Streams

Object Streams

- The Java language supports a very general mechanism called *object serialization*, that makes it possible to write any object to a stream and read it again later.

- To *serialize* an object in an output file, simply instantiate an appropriate `ObjectOutputStream` object and call its `writeObject` method:

```
BankAccount acc = ...;
ObjectOutputStream oos =
  new ObjectOutputStream(new FileOutputStream("accounts.ser"));
oos.writeObject(acc);
```

Object Streams

- To read the object back in, instantiate an `ObjectInputStream` object and call its `readObject` method:

```
ObjectInputStream ois =
    new ObjectInputStream(new FileInputStream("accounts.ser"));
BankAccount acc = (BankAccount)ois.readObject();
```

- To be serialized, the object's class must implement the (empty) interface `Serializable`.

```
public class BankAccount implements Serializable {...}
```

- Note: Arrays are serializable.

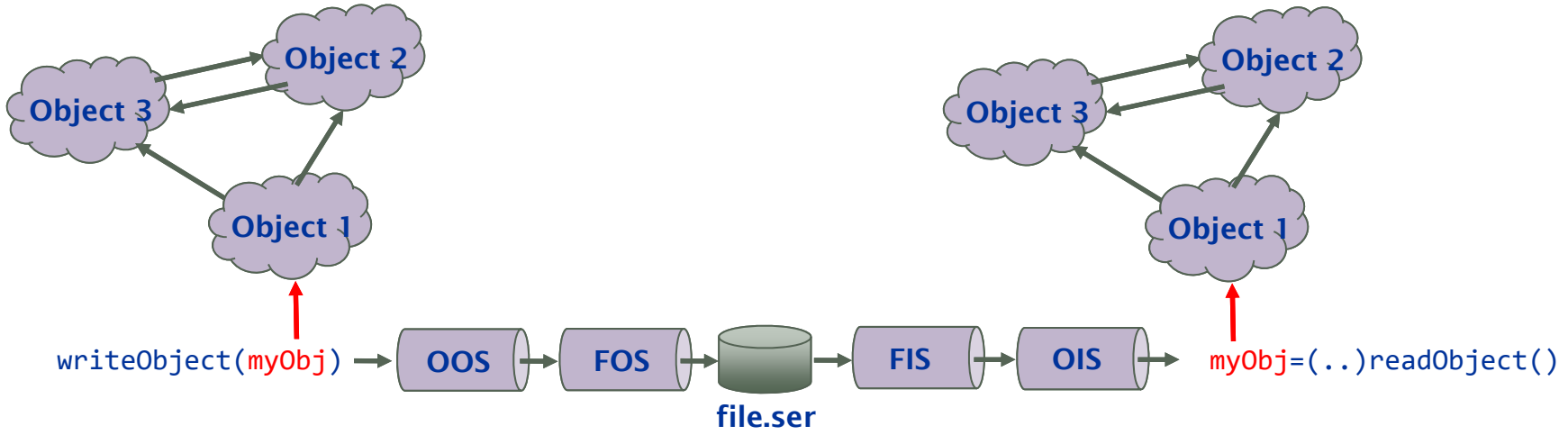
Object Streams

- During the serialization process the content of all instance variables of the object are saved in the `ObjectOutputStream`.

- If the object contains references to other objects, *these objects will be serialized, too* (insofar they are also serializable)
  - For example, if a list or an array of bank accounts is serialized:
    - all bank account objects contained in the list or the array, as well as
    - all the objects (directly or indirectly) referenced by these objects will be serialized.

Object Streams

- The serialization process does not modify the objects graph, i.e. the existing references between the objects are not lost and no object is wrongly duplicated.



Object Streams

- Example – Serializing two arrays containing two bank accounts :

```
BankAccount acc1 = new BankAccount(1000); // Only..
BankAccount acc2 = new BankAccount(1000); // .. two objects
BankAccount[] accounts1 = {acc1, acc2, acc1};
BankAccount[] accounts2 = {acc2, acc1, acc1};
System.out.println("Out 1:" + Arrays.toString(accounts1));
System.out.println("Out 2:" + Arrays.toString(accounts2));

ObjectOutputStream out =
  new ObjectOutputStream(new FileOutputStream("accounts.ser"));
out.writeObject(accounts1);
out.writeObject(accounts2);
...
```

Object Streams

- Getting the bank accounts back:

```
ObjectInputStream in =
  new ObjectInputStream(new FileInputStream("accounts.ser"));
accounts1 = (BankAccount[])in.readObject();
accounts2 = (BankAccount[])in.readObject();
System.out.println("In 1:" + Arrays.toString(accounts1));
System.out.println("In 2:" + Arrays.toString(accounts2));
```

- Output:
  - The correct object graph has been reconstituted

```
Out 1:[BankAccount@503bbcfd, BankAccount@1f4af32, BankAccount@503bbcfd]
Out 2:[BankAccount@1f4af32, BankAccount@503bbcfd, BankAccount@503bbcfd]
In 1:[BankAccount@39890510, BankAccount@52ab7af2, BankAccount@39890510]
In 2:[BankAccount@52ab7af2, BankAccount@39890510, BankAccount@39890510]
```

Object Streams

- If an object *may* be serialized, it does not mean that it *can* be serialized. The serialization process will abort if:
  - A non-serializable superclass of the object's class has no no-arg. constructor.
    - In that case, a no-arg. constructor is invoked when the object is deserialized, in order to restore the full object state. If it is missing, the object cannot be deserialized.
  - A `NotSerializableException` is thrown.
    - Doing this, it is possible to prevent, in a subclass, the serialization of an object even if one of its superclasses implements `Serializable`.
    - Note that, to generate an exception during the serialization process, you have to interfere with the serialization mechanism (see below).

Object Streams

- The object being serialized references a non-serializable object
  - You can avoid this by declaring the reference variable as **`transient`**. Transient objects are skipped by the serialization process.
  - Example: To be able to serialize the following class, the reference to the (non-serializable) `PrintWriter` object must be made transient.

```
public class BankAccount implements Serializable {
   private double balance;
   private transient PrintWriter out; // "out" is not serializable
   ...
}
```

- Note that static variables are ignored by the serialization process (they are not part of the object).

Object Streams

- Versioning
  - When an object is serialized, the state of the object and the name of the object's class(es) are stored, but not the whole class byte codes.
  - Therefore, there is no guarantee that the serialized object will be deserialized using the same class. It could be a another version, for example:
    - if some time elapsed between the serialization and the deserialization of the object, and/or
    - if the object was moved to another location through Internet
  - For that purpose, a class can indicate if it is *compatible* with the original version of itself, by using a **`long`** number called *Serial version Unique IDentifier* (SUID).

Object Streams

- The SUID represents a "fingerprint" of the class, obtained by selecting relevant information about the class and applying a special hash algorithm (Secure Hash Algorithm: SHA) on it.
- Two classes are compatible, only if the SUID stored in the object stream during the serialization step is equal to the SUID of the current class. This is the condition for an object to be deserialized.
- However, you can deactivate the default calculation of the SUID, by specifying your own SUID. When a class had a static data field named `serialVersionUID`, it will not compute the "fingerprint" manually but will use that value instead, allowing the programmer to control the class compatibilities.

```
private static final long serialVersionUID = ...; // User SUID
```

Object Streams

- You can obtain the SUID of a (serializable) class by calling the JDK utility **`serialver`** (a GUI version exists, too).
  - Example: obtaining the SUID for the class **`String:`**
    - Input

```
serialver java.lang.String
```

    - Output

```
java.lang.String: private static final long serialVersionUID = -6849794470754667710L;
```

- That utility is automatically called in Eclipse, when needed.

Object Streams

- Simple Serialization example:
  - The following program demonstrates the serialization of a Bank object, that contains a few bank accounts in an array list. If a file with serialized data exists, then it is loaded. Otherwise the program starts with a new bank.
  - Note that the `Bank` and `BankAccount` classes, both implement the `Serializable` interface.

BankAccount.java

```
import java.io.*;
/**
   A bank account has a balance that can be changed by deposits and withdrawals.
*/
public class BankAccount implements Serializable {
  private int accountNumber;
  private String owner;
  private double balance;

  /**
     Constructs a bank account with a given balance
     @param anAccountNumber the account number for this account
     @param anOwner the owner for this account
     @param initialBalance the initial balance
  */
  public BankAccount(int anAccountNumber, String anOwner, double initialBalance) {
    accountNumber = anAccountNumber;
    owner = anOwner;
    balance = initialBalance;
  }

  /**
     Gets the account number of this bank account.
     @return the account number
  */
  public int getAccountNumber() {   return accountNumber; }
```

BankAccount.java *(cont.)*

```
 /**
    Gets the owner of this bank account.
    @return the account owner
 */
 public int getOwner() { return owner; }

 /**
    Deposits money into the bank account.
    @param amount the amount to deposit
 */
 public void deposit(double amount) { balance += amount; }

 /**
    Withdraws money from the bank account.
    @param amount the amount to withdraw
 */
 public void withdraw(double amount) { balance -= amount; }

 /**
    Gets the current balance of the bank account.
    @return the current balance
 */
 public double getBalance() { return balance; }
}
```

Bank.java

```
import java.io.*;
import java.util.*;
/**
   This bank contains a collection of bank accounts.
*/
public class Bank implements Serializable {

  private ArrayList<BankAccount> accounts;

  /**
     Constructs a bank with no bank accounts.
  */
   public Bank() {
     accounts = new ArrayList<>();
   }

  /**
     Adds an account to this bank.
     @param acc the account to add
  */
   public void addAccount(BankAccount acc) {
     accounts.add(acc);
   }
```

Bank.java *(cont.)*

```
   /**
      Finds a bank account with a given number.
      @param accountNumber the number to find
      @return the account with the given number, or null if there
      is no such account
   */
   public BankAccount find(int accountNumber) {
     for (BankAccount acc : accounts)
       if (acc.getAccountNumber() == accountNumber) // Found a match
         return acc;
     return null; // No match in the entire array list
   }
}
```

SerialDemo.java

```
import java.io.*;
/**
   This program demonstrates serialization of a Bank object.
   Bank accounts are added to the bank. Then the bank
   object is saved.
*/
public class SerialDemo {

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

    Bank firstBankOfJava;

    String filename = "bank.ser";
    if (Files.exists(Paths.get(filename)))
      try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename))){
        firstBankOfJava = (Bank)in.readObject();
      }
    else {
      firstBankOfJava = new Bank();
      firstBankOfJava.addAccount(new BankAccount(1001, "E. Dijkstra", 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, "E. Gamma", 10000));
    }
```

SerialDemo.java *(cont.)*

```
    // Deposit some money in one account and print the accounts.
    BankAccount acc = firstBankOfJava.find(1001);
    acc.deposit(100);
    System.out.println(acc.getAccountNumber() + ": " + acc.getOwner() +
                       " - " + acc.getBalance());

    acc = firstBankOfJava.find(1015);
      System.out.println(acc.getAccountNumber() + ": " + acc.getOwner() +
                         " - " + acc.getBalance());

    try (ObjectOutputStream out =
                        new ObjectOutputStream(new FileOutputStream(filename))) {
      out.writeObject(firstBankOfJava);
    }
  }
}
```

SerialDemo.java

- Program Run

```
1001:20100.0
1015:10000.0
```

- Second Program Run

```
1001:20200.0
1015:10000.0
```

Object Streams

- Object streams methods:
  - Besides `writeObject` and `readObject`, the object stream classes also have other methods, in particular to serialize/deserialize primitive data types. These methods have identical names as by the `DataOutputStream` and `DataInputStream` classes

```
ObjectOutputStream out = new ObjectOutputStream(
                                  new FileOutputStream("file.ser");
out.writeInt(12345);
out.writeDouble(123.45);
...
ObjectInputStream in = new ObjectIntputStream(
                                   new FileInputStream("file.ser");
int i = in.readInt();
double d = in.readDouble();
```

Object Streams

- Customizing serialization process:
  - You can customize the default Java serialization process, by "overriding" the `writeObject` and `readObject` methods with `private`(!) methods in the serializable class.
  - Overriding the serialization process can be sometimes very useful. For example to serialize attributes of non-serializable objects or to "hide" data inside the serialized stream.
  - Example: Obscuring the account number of a bank account during the serialization process, so that this sensitive data cannot be directly accessible from the serialization stream.
    - To obscure the account number, we convert it to a string and encrypt it with the "Caesar cipher" algorithm.
    - For that, we define in the `BankAccount` class convenient `writeObject` and `readObject` methods, and declare `accountNumber` as `transient`.

Object Streams

```
public class BankAccount implements Serializable {
  private static final int KEY = 5; // Encryption key
  private transient int accountNumber;
  private String owner;
  private double balance;
  ...
  private void writeObject(ObjectOutputStream out) throws IOException {
    // Convert the account number into a char array
    char[] array = ("" + accountNumber).toCharArray();
    out.writeInt(array.length); // Write the account number length
    // Encrypt and write each account number character
    for (char ch : array) out.writeChar(encrypt(ch, KEY));

    // Write all non-static and non-transient fields of this BankAccount
    out.defaultWriteObject();
  }
```

Object Streams

```
private void readObject(ObjectInputStream in)
                    throws IOException, ClassNotFoundException {
  int numberLength = in.readInt(); // Read the account number length
  // Read and decrypt each number character, convert them to int,
  // and rebuild the account number
  accountNumber = 0;
  for (int i = 0; i < numberLength; i++) {
    int nextDigit =
        Integer.parseInt("" + (char)encrypt(in.readChar(), -KEY));
    accountNumber = accountNumber * 10 + nextDigit;
  }

  // Read all non-static and non-transient fields from the BankAccount
  in.defaultReadObject();
}
```

Object Streams

```
  private int encrypt(int x, int key) {
    return (x + key) % 256;  // Encrypt a value and return it
  }
}
```

- Note that this example only assumes that you want to obscure some serialized data using a very simple algorithm. Java offers other tools to encrypt data with a high level of security without having to stress about the details of the encryption.