Date and Time, I/O Streams

IT Academy

softserve

Agenda

- Java 7 Date Time API
- Java 8 Date Time API
- Typical Uses of I/O System
- What Is a Stream?
- Byte Oriented Streams
- Character-Oriented Streams
- Buffered Input / Output

Java 7 Date Time API

softserve

Backstory

- The **`Date`** class was the work of James Gosling and Arthur van Hoff.
- Added in JDK 1.0, mostly deprecated in JDK 1.1, never removed.
- IBM donated **`Calendar`** class code to Sun.

The `Date` class

- The **`Date`** class available in **`java.util`** package, this class encapsulates the ***current date*** and ***time***.
- The **`Date`** class supports two constructors.

- **`Date()`**

  This constructor initializes the object with the ***current date*** and ***time***.

- **`Date(long millisec)`**

  This constructor accepts an argument that equals the ***number of milliseconds*** that have elapsed since **midnight, January 1, 1970**.

Getting Current Date and Time

- The **`Date`** object stores information about current date and time.

```
// Instantiate a Date object
Date date = new Date();

// Display time and date using toString()
System.out.println(date);
```

```
Fri Feb 07 09:40:51 EET 2020
```

- Methods **`getTime()`** and **`System.currentTimeMillis()`** return the current date and time as milliseconds since ***January 1st 1970***.

```
Date date = new Date();
long currentTime1 = date.getTime();
System.out.println(currentTime1);

long currentTime2 = System.currentTimeMillis();
System.out.println(currentTime2);
```

```
1581061806149
1581061806149
```

softserve

Useful Methods of the `Date` class

- **`long getTime()`**

  Returns the number of milliseconds that have elapsed since January 1, 1970.

- **`void setTime(long time)`**

  Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970.

- **`boolean after(Date date)`**

  Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false.

- **`boolean before(Date date)`**

  Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false.

- **`int compareTo(Date date)`**

  Compares the value of the invoking object with that of date.

Problems Getting a `Date` class

- Conceptually an instant, not a date
- Properties have random offsets:
  - *Some **zero-based**, like **month** and **hours***
  - *Some o**ne-based**, like **day of the month***
  - ***Year** has an **offset of 1900***
- Mutable, not thread-safe
- Not internationalizable
- Millisecond granularity
- Does not reflect UTC

The `GregorianCalendar` Class

```
Calendar calendar = new GregorianCalendar();

//set date to 05.02.2020 15:37
calendar.set(Calendar.YEAR, 2020);
calendar.set(Calendar.MONTH, 1); // 1 = February
calendar.set(Calendar.DAY_OF_MONTH, 5);
calendar.set(Calendar.HOUR_OF_DAY, 15); // 24 hour
clock
calendar.set(Calendar.MINUTE, 37);

// get date components
int year = calendar.get(Calendar.YEAR);
int month = calendar.get(Calendar.MONTH);
int dayOfMonth = calendar.get(Calendar.DAY_OF_MONTH);
int hourOfDay  = calendar.get(Calendar.HOUR_OF_DAY);
int minute = calendar.get(Calendar.MINUTE);
int second = calendar.get(Calendar.SECOND);

//add one day
calendar.add(Calendar.DAY_OF_MONTH, 1);

Date date = calendar.getTime();

System.out.println(date);
```

```
Thu Feb 06 15:37:27 EET
2020
```

- The **`GregorianCalendar`** is a concrete implementation of a **`Calendar`** class that implements the ***normal Gregorian calendar*** with which you are familiar.
- The **`getInstance()`** method of **`Calendar`** returns a **`GregorianCalendar`** initialized with the ***current date and time*** in the default locale and time zone.

softserve

Useful Methods of the `Calendar` class

- **`void add(int field, int amount)`**

  Adds the specified (signed) amount of time to the given time field, based on the calendar's rules.

- **`int get(int field)`**

  Gets the value for a given time field.

- **`Date getTime()`**

  Gets this Calendar's current time.

- **`long getTimeInMillis()`**

  Gets this Calendar's current time as a long.

- **`boolean isLeapYear(int year)`**

  Determines if the given year is a leap year.

Useful Methods of the `Calendar` class

- **`void set(int field, int value)`**
  Sets the time field with the given value.
- **`void set(int year, int month, int date)`**
  Sets the values for the fields year, month, and date.
- **`void setTime(Date date)`**
  Sets this Calendar's current time with the given Date.
- **`void setTimeInMillis(long millis)`**
  Sets this Calendar's current time from the given long value.
- **`void setGregorianChange(Date date)`**

  Sets the GregorianCalendar change date.

Problems Getting a `Calendar` class

- Conceptually an instant, not a calendar.
- Zero-based offsets
- Stores internal state in two different ways
  - *milliseconds from epoch*
  - *set of fields*
- Has bugs and performance issues
- Mutable, not thread-safe

Java 8 Date Time API

softserve

Backstory

- 2002 - Stephen Colebourne starts open source Joda-Time project
- 2005 - Release of Joda-Time 1.0
- 2007 - JSR 310, for inclusion in Java
- 2011 - Release of Joda-Time 2.0
- 2014 - Finally, the date and time API is in Java 8

New Packages

- **`java.time`**
  instants, durations, dates, times, time zones, periods.
- **`java.time.format`**
  formatting and parsing.
- **`java.time.temporal`**
  field, unit, or adjustment access to temporals.
- **`java.time.zone`**
  support for time zones.
- **`java.time.chrono`**
  calendar systems other than ISO-8601.

Design Principles

- Distinguish between machine and human views.
- Well-defined and clear purpose.
- Immutable, thread-safe.
- Reject `null` and bad arguments early.
- Extensible, by use of strategy pattern.
- Fluent interface with chained methods.

Commonly Used Classes

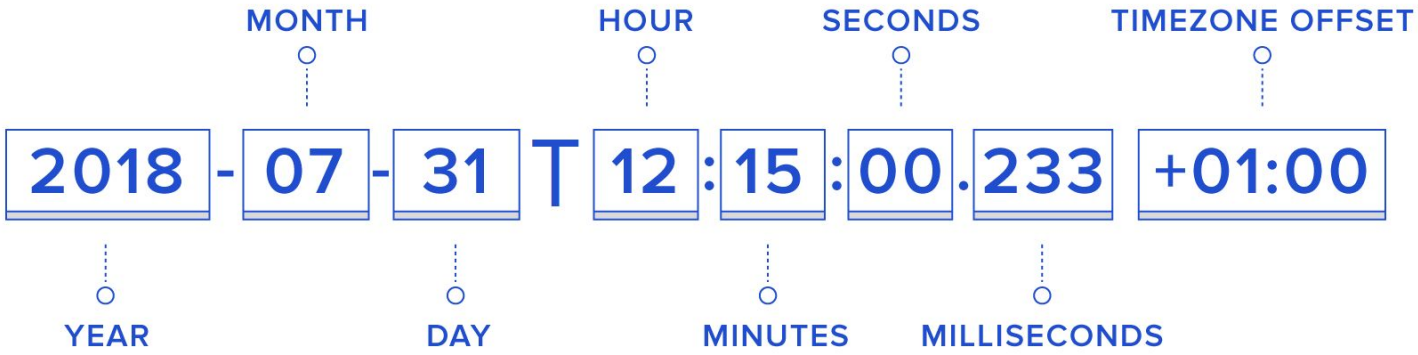
- **`LocalDate`**
  - ISO 8601 ***date without time zone*** and ***time***
  - Corresponds to SQL DATE type
- **`LocalTime`**
  - ISO 8601 ***time without time zone*** and ***date***
  - Corresponds to SQL TIME type
- **`LocalDateTime`**
  - ISO 8601 ***date*** and ***time without time zone***
  - Corresponds to SQL TIMESTAMP type
- **`DateTimeFormatter`**

  Formatter for displaying and parsing date-time objects

softserve

The ISO 8601 Standard

- The **International standard** for representation of ***dates*** and ***times***.
- Uses the ***Gregorian calendar system***.
- Ordered from most to least significant: *year*, *month*, *day*, *hour*, *minute*.
- Each date and time value has a ***fixed number of digits with leading zeros***.
- Uses four-digit year at minimum, **YYYY**.



softserve

The `LocalDate` Class

- Obtain a **`LocalDate`** object corresponding to the ***local date of today***.

```
LocalDate localDateNow = LocalDate.now();
System.out.println(localDateNow);
```

```
2020-02-07
```

- Create a **`LocalDate`** object from ***year***, ***month*** and ***day*** information.

```
LocalDate localDate = LocalDate.of(2020, Month.MARCH, 16);
System.out.println(localDate);
```

```
2020-03-16
```

The `LocalDate` Class

- Access the date information of a **`LocalDate`** object.

```
int year = localDate.getYear();
Month month = localDate.getMonth();
int dayOfMonth = localDate.getDayOfMonth();
int dayOfYear = localDate.getDayOfYear();
DayOfWeek dayOfWeek = localDate.getDayOfWeek();

System.out.println("Year: " + year);
System.out.println("Month: " + month);
System.out.println("Day of Month: " + dayOfMonth);
System.out.println("Day of Year: " + dayOfYear);
System.out.println("Day of Week: " + dayOfWeek);
```

```
Year: 2020
Month: MARCH
Day of Month: 16
Day of Year: 76
Day of Week: MONDAY
```

The `LocalDate` Class

- Simple date calculations with the **`LocalDate`** object.

```
LocalDate localDate = LocalDate.of(2020, Month.MARCH, 16);
System.out.println(localDate);

localDate = localDate.minusYears(2);
localDate = localDate.plusMonths(5);
localDate = localDate.minusDays(132);
localDate = localDate.plusWeeks(4);
System.out.println(localDate);
```

```
2020-03-16
```

```
2018-05-04
```

softserve

The `LocalTime` Class

- Create a **`LocalTime`** object that represents the ***exact time of now***.

```
LocalTime localTimeNow = LocalTine.now();
System.out.println(localTimeNow);
```

```
12:10:18.625
```

- Create a **`LocalTime`** object from a specific ***amount of hours***, ***minutes***, ***seconds*** and ***nanoseconds***.

```
LocalTime localTime = LocalTime.of(15, 30, 25, 845);
System.out.println(localTime);
```

```
15:30:25.000000855
```

softserve

The `LocalTime` Class

- Access the time information of a **`LocalTime`** object.

```
int hour = localTime.getHour();
int minute = localTime.getMinute();
int second = localTime.getSecond();
int nanoOfSeconds = localTime.getNano();

System.out.println("Hour: " + hour);
System.out.println("Minute: " + minute);
System.out.println("Second: " + second);
System.out.println("Nano of Seconds: " + nanoOfSeconds);
```

```
Hour: 15
Minute: 30
Second: 25
Nano of Seconds: 845
```

The `LocalTime` Class

- Simple time calculations with the **`LocalTime`** object.

```
LocalTime localTime = LocalTime.of(15, 30, 25, 845);
System.out.println(localTime);

localTime = localTime.minusHours(27);
localTime = localTime.plusMinutes(179);
localTime = localTime.minusSeconds(683);
localTime = localTime.plusNanos(8345);
System.out.println(localTime);
```

```
15:30:25.000000845
```

```
15:18:02.000009190
```

softserve

The LocalDateTime Class

- The **`LocalDateTime`** class represents a local date and time ***without any time zone information***.
- You could view the **`LocalDateTime`** as a combination of the **`LocalDate`** and **`LocalTime`** classes.

```
LocalDateTime localDateTime = LocalDateTime
        .of(2020, Month.MARCH, 16, 15, 30, 25, 845);

System.out.println(localDateTime);
```

```
2020-03-16T15:30:25.000000845
```

softserve

The `DateTimeFormatter` Class

- The **`DateTimeFormatter`** class is used to ***parse*** and ***format dates*** represented with the classes in the Java 8 date time API.
- The **`DateTimeFormatter`** class contains a ***set of predefined constant*** which can ***parse*** and ***format dates*** from ***standard date formats***.
- Each of these predefined **`DateTimeFormatter`** instances are ***preconfigured*** to format and ***parse dates to / from different formats***.

```
// Some of DateTimeFormatter constant
DateTimeFormatter.BASIC_ISO_DATE

DateTimeFormatter.ISO_LOCAL_DATE
DateTimeFormatter.ISO_LOCAL_TIME
DateTimeFormatter.ISO_LOCAL_DATE_TIME

DateTimeFormatter.ISO_DATE
DateTimeFormatter.ISO_TIME
DateTimeFormatter.ISO_DATE_TIME

DateTimeFormatter.ISO_WEEK_DATE
DateTimeFormatter.ISO_ZONED_DATE_TIME
DateTimeFormatter.ISO_INSTANT
```

The `DateTimeFormatter` Class

- The **`format()`** method is declared on both the ***formatter objects*** and the ***date/time objects***.

```
LocalDateTime localDateTime = LocalDateTime.now();
System.out.println(localDateTime);

String basicIsoDate = localDateTime.format(
        DateTimeFormatter.BASIC_ISO_DATE);
String isoDateTime = localDateTime.format(
        DateTimeFormatter.ISO_DATE_TIME);
String isoLocalDateTime = localDateTime.format(
       DateTimeFormatter.ISO_LOCAL_DATE_TIME);
String isoWeekDate = localDateTime.format(
       DateTimeFormatter.ISO_WEEK_DATE);
```

```
2020-02-07T13:16:46.454
20200207
2020-02-07T13:16:46.454
2020-02-07T13:16:46.454
2020-W06-5
```

Others useful Classes and Interfaces

- **`Temporal`**
  Basic interface for **`DateTime`** classes
  `LocalDate` / **`LocalTime`** / **`LocalDateTime`** ...
- **`Instant`**
  Start of nanoseconds in timeline. Useful for timestamp
- **`Clock`**
  Allowing **`Temporal`** creation with alternate clock
- **`TemporalAmount`**
  Basic interface for classes that represent amount of time
  `Duration` / **`Period`**

The `Instant` Class

- Point on a discretized time-line
- Stored to nanosecond resolution

  `long` *for seconds since epoch, and*

  `int` *for nanosecond of second*
- Convert to any date time field using a **`Chronology`**
- Use for event time-stamps

The `Clock` Class

- Gets the current instant using a time-zone

  Use instead of **`System.currentTimeMillis()`**

  Use an alternate clock for testing **`Clock`**

```
ZoneId zoneId = ZoneId.of("Europe/Kiev");
ZonedDateTime localDateTime = ZonedDateTime.now(Clock.system(zoneId));

System.out.println(localDateTime);
```

```
2020-02-07T13:48:25.821+02:00[Europe/Kiev]
```

softserve

The `Period` Class

- A length of elapsed time. Defined using calendar fields

  years, months, and days (not minutes and seconds).

- Takes time zones into account for calculation.

```
LocalDateTime localDateTime = LocalDateTime.now();
String dateTime = localDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM));
System.out.println(dateTime);

Period period = Period.of(2, 7, 15);
localDateTime = localDateTime.plus(period);

dateTime = localDateTime.format(
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM));
System.out.println(dateTime);
```

```
7 лют. 2020 14:00:56
22 вер. 2022 14:00:56
```

softserve

The `Duration` and `Chronology`

- The **`Duration`** class
  - Precise length of elapsed time, in nanoseconds
  - Does not use date-based constructs like years, months, and days
  - Can be negative, if end is before start
- The **`Chronology`** interface
  - Pluggable calendar system
  - Provides access to date and time fields
  - Built-in
    - ISO8601 (default): **`IsoChronology`**
    - Chinese: **`MinguoChronology`**
    - Japanese: **`JapaneseChronology`**
    - Thai Buddhist: **`ThaiBuddhistChronology`**
    - Islamic: **`HijrahChronology`**

softserve

Staying Constant

- Day of week, for example

```
DayOfWeek.FRIDAY
```

- Month , for example

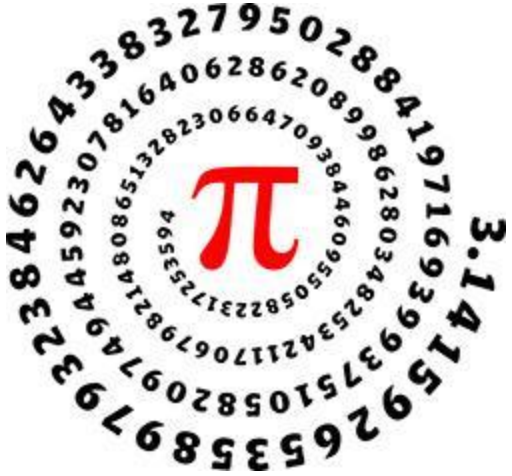
```
Month.MAY
```

- Time units, for example

```
ChronoUnit.DAYS
```

- Other useful constants:

```
LocalTime.MIDNIGHT   // 00:00
LocalTime.NOON       // 12:00
```



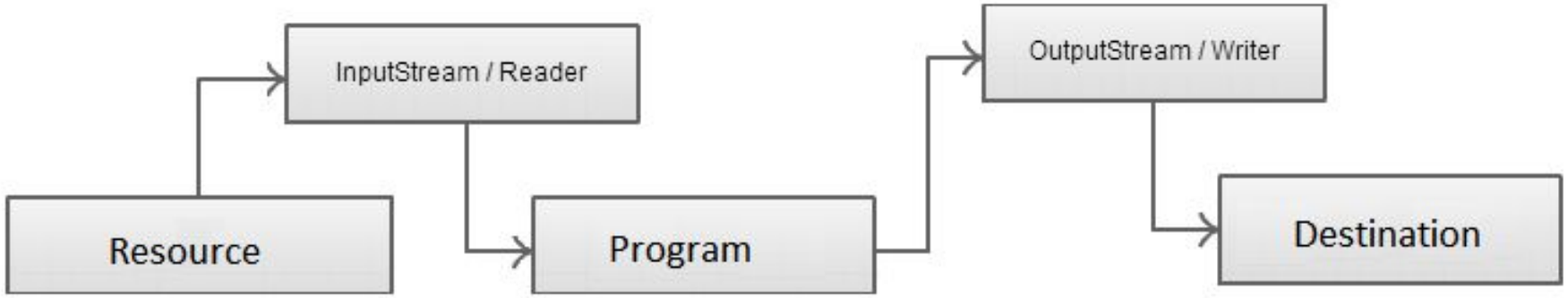
softserve

Input/Output Streams API

softserve

What Is a Stream?

- A **stream** is an ***ordered sequence of bytes*** of undetermined length.
- **Input streams** move bytes of data ***from some generally external source to Java program***.
- **Output streams** move bytes of data ***from Java program to some generally external source***.

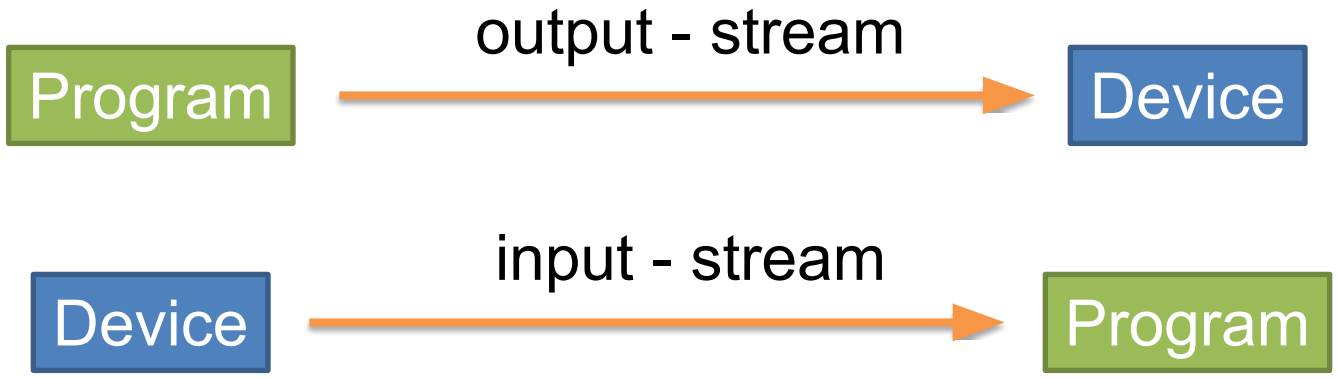


softserve

Streams

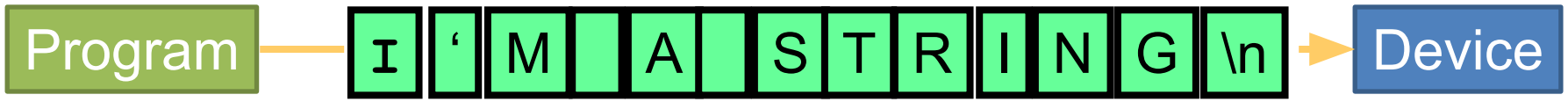
- Usual Purpose:

  Storing data to «nonvolatile» devices, e.g. harddisk.
- Classes provided by package **`java.io`**.
- Data is ***transferred*** to devices by **streams**

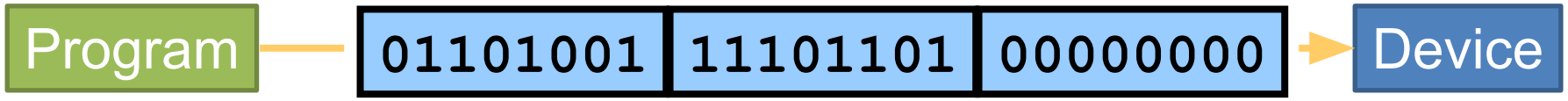


Streams

- JAVA distinguishes between **2 types** of **streams**:
  - **Text Streams**, containing "*characters*"



  - **Binary Streams**, containing **8** *bit information*



softserve

Streams

- Streams in JAVA are ***Objects***, of course!
  - **2 *types*** of streams (**text** / **binary**) and
  - **2 *directions*** (**inpu**t / **output**)

- Results in ***4 base-classes*** dealing with **I/O**:
  1. **`Reader`** : text-input
  2. **`Writer`** : text-output
  3. **`InputStream`** : byte-input
  4. **`OutputStream`** : byte-output

softserve

The `File` Class

- A **`File`** object can refer to ***either a file***:

```
File file = new File("data.txt");
```

- or a ***directory***:

```
File file = new File("D:\\development");
```

The `File` Class

```
String dirName = "D:\\Development";
String fileName = "data.txt";

File newDirectory = new File(dirName);
boolean isDirCreated = newDirectory.mkdir();

if (isDirCreated) {
    File newFile = new File(dirName + File.separator + fileName);
    boolean isFileCreated = newFile.createNewFile();

    if (isFileCreated) {
        System.out.println(newFile.getCanonicalPath());
    }
}
```

Useful Methods of the `File` class

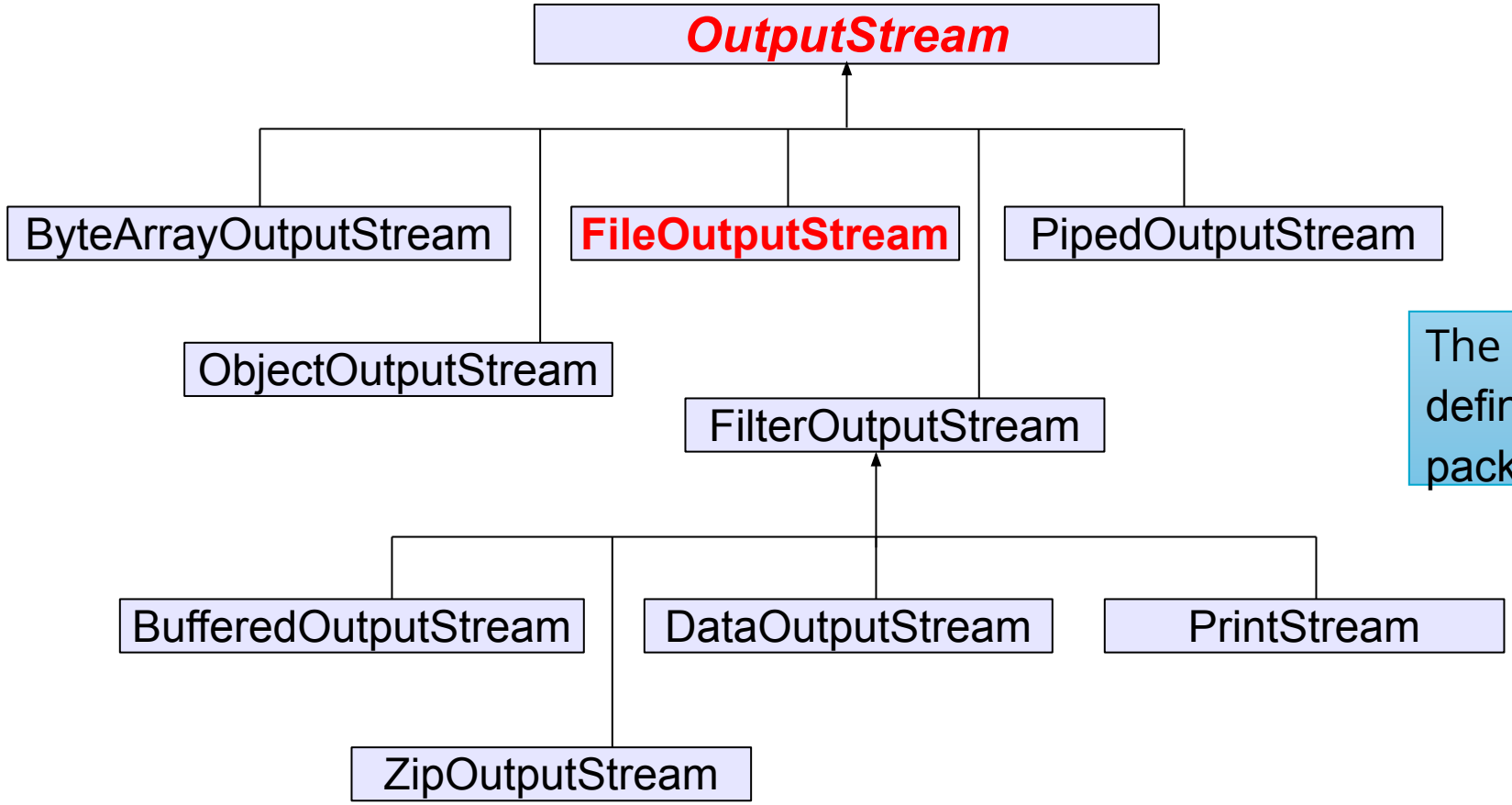
- **`isFile()` / `isDirectory()`**
  Returns **`true`** if and only if the file denoted by this abstract pathname is a file (directory).
- **`canRead()`**
  Returns **`true`**, if the specified file exists its path name and the file is allowed to be read by the application.
- **`canWrite()`**
  Returns **`true`** if the application to write to the file, else the method returns **`false`.**
- **`length()`**
  Length of the file in bytes (**`long`**) or **`0`** if nonexistent.
- **`list()`**
  If the **`File`** object is a directory, returns a **`String`** array of all the files and directories contained in the directory; otherwise, **`null`.**
- **`mkdir()`**
  Creates a new subdirectory.
- **`delete()`**
  Deletes the directory or file and returns **`true`** if successful.

Binary Files

- Stores ***binary images of information*** identical to the binary images stored in main memory.
- Example: *writing of the integer* '**42**'
  - Text-File: "**42**" (internally translated to 2 **16-bit** representations of the characters '**4**' and '**2**')
  - Binary-File: **00101010**, one byte (= **42** decimal)

Byte-Oriented Output Stream Classes

- The following is the ***byte-oriented output stream*** class hierarchy:



The **`ZipOutputStream`** is defined in **`java.util.zip`** package

Methods of `OutputStream` Class

- Writing data:
  - **`write(...)`** methods write data to the stream. Written data is buffered.
  - Use **`flush()`** method to flush any buffered data from the stream.
  - throws **`IOException`** if an I/O error occurs.
- There are **3** main write methods:
  - **`void write(int data)`**

    Writes a single character (even though data is an integer, data must be set such that: **`0 <= data <= 255`**)
  - **`void write(byte[] buffer)`**

    Writes all the bytes contained in buffer to the stream
  - **`void write(byte[] buffer, int offset, int length)`**

    Writes length bytes to stream starting from **`buffer[offset]`**

Methods of `OutputStream` Class

- **`flush()`**
  - To improve performance, almost all output protocols buffer output.
  - Data written to a stream is not actually sent until buffering thresholds are met.
  - Invoking **`flush()`** causes the **`OutputStream`** to clear its internal buffers.
- **`close()`**

  Closes stream and releases any system resources.

Creating a `FileOutputStream` Instance

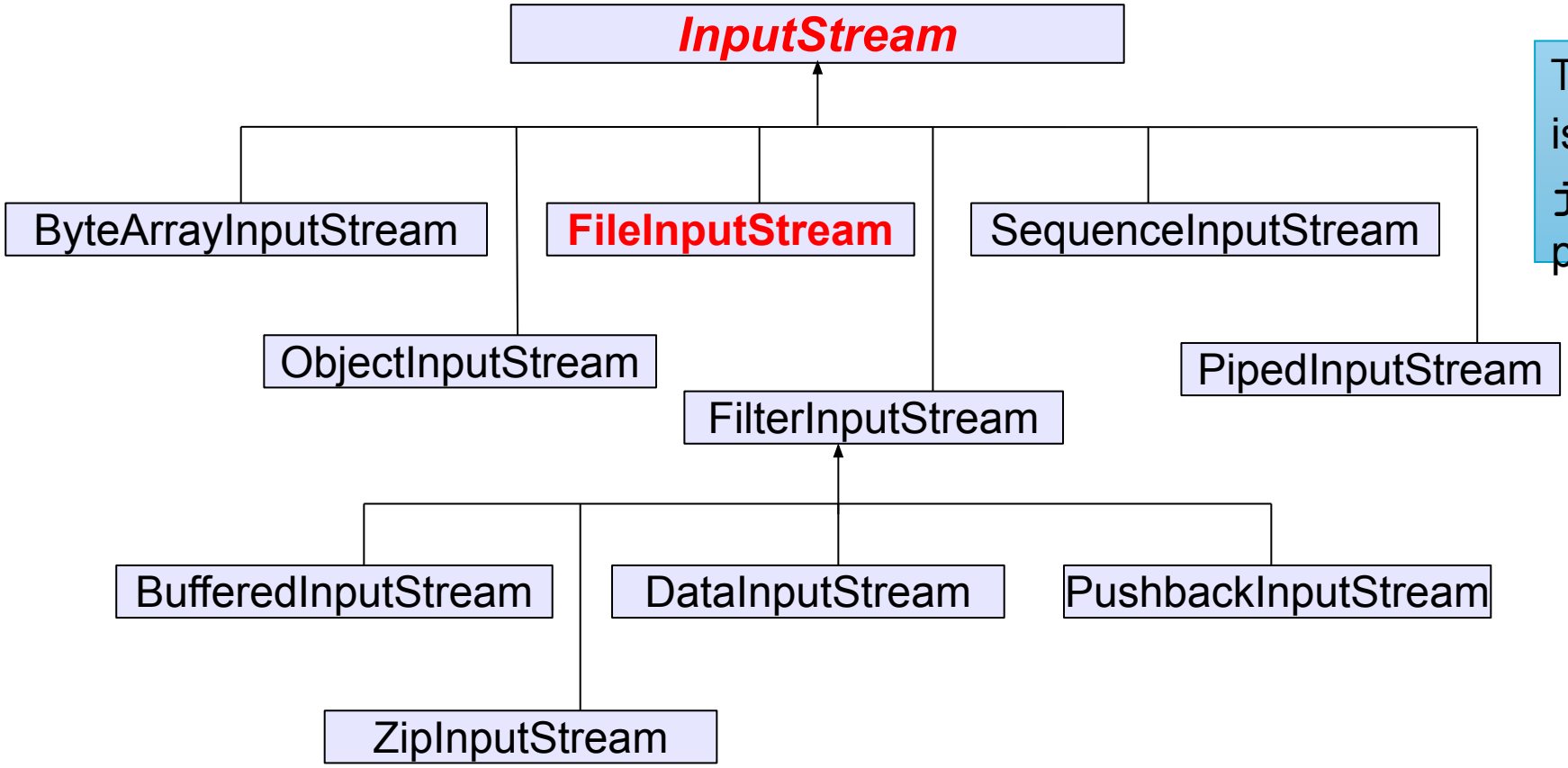
```
String fileName = "data.txt";

String str = "Hello I/O!";
byte[] wData = str.getBytes();

try (FileOutputStream fileOutputStream =
                    new FileOutputStream(fileName /*, true */)) {

    fileOutputStream.write(wData);
    System.out.println("Was wrote " + wData.length + " bytes");

} catch (IOException e) {
    e.printStackTrace();
}
```

```
Was wrote 10 bytes
```

softserve

Byte-Oriented Input Stream Classes

- The following is the ***byte-oriented input stream*** class hierarchy:



The **`ZipInputStream`** is defined in **`java.util.zip`** package

Methods of `InpupStream` Class

- Reading data:
  - **`read()`** methods will block until data is available to be read and return the number of bytes read.
  - **`-1`** is returned if the Stream has ended.
  - throws **`IOException`** if an I/O error occurs.
- There are 3 main read methods:
  - **`int read()`**

    Reads a single byte. Returns it as integer.
  - **`int read(byte[] buffer)`**

    Reads bytes and places them into buffer. Returns the number of bytes read.
  - **`int read(byte[] buffer, int offset, int length)`**

    Reads up to length bytes and places them into buffer. First byte read is stored in **`buffer[offset]`**. Returns the number of bytes read.

Methods of `InpupStream` Class

- **`available()`**

  Returns the number of bytes which can be read without blocking.

- **`skip(long n)`**

  Skips over a number of bytes in the input stream.

- **`close()`**

  Closes stream and release any system resources.

Creating a `FileInputStream` Instance

```
byte[] rData = new byte[15];

try (FileInputStream fileInputStream = new FileInputStream(fileName)) {

    int byteAvailable = fileInputStream.available();
    int byteCount = fileInputStream.read(rData, 0, byteAvailable);

    System.out.println("Was read " + byteCount + " bytes");
    System.out.println(Arrays.toString(rData));
    System.out.println(new String(rData));

} catch (IOException e) {
    e.printStackTrace();
}
```

```
Was read 10 bytes
[72, 101, 108, 108, 111, 32, 73, 47, 79, 33, 0, 0, 0, 0, 0]
Hello I/O!
```

softserve

Creating a `FileInputStream` Instance

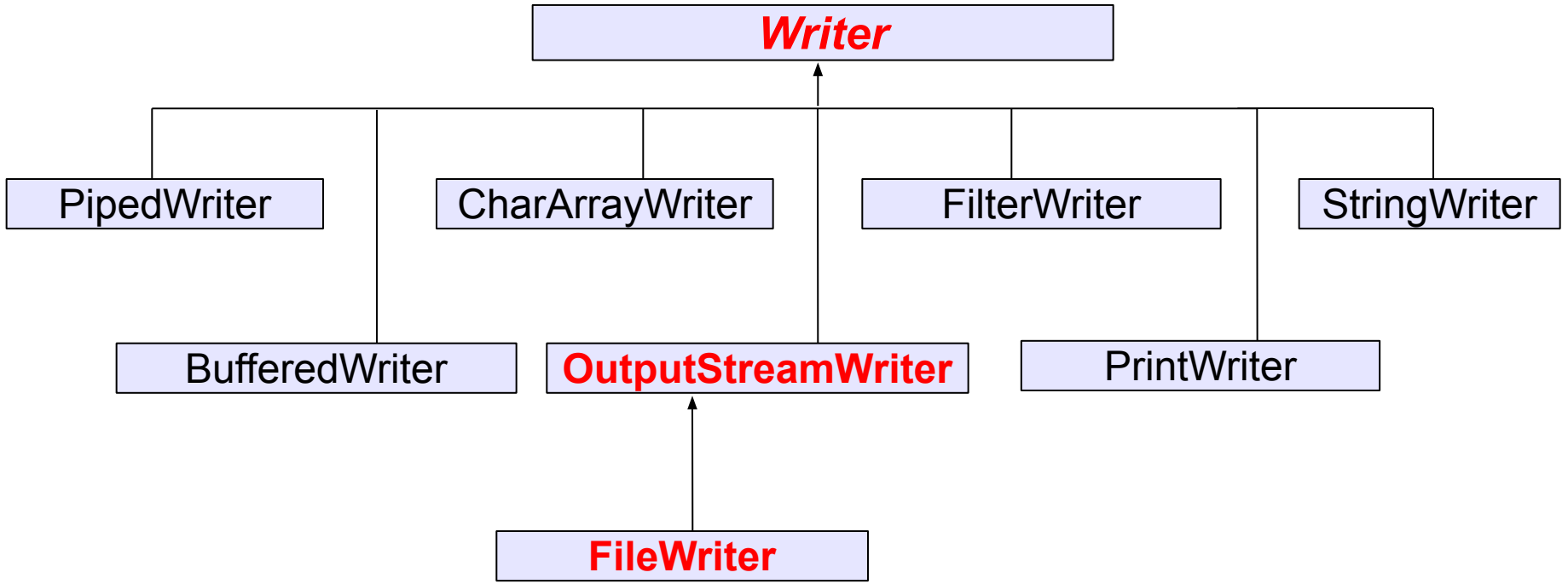
```
String fileName = "data.txt";

try (FileInputStream fileInputStream = new FileInputStream(fileName)) {

    int b = 0;

    while ((b = fileInputStream.read()) != -1) {
        System.out.print( (char)b );
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

```
Hello I/O!
```

softserve

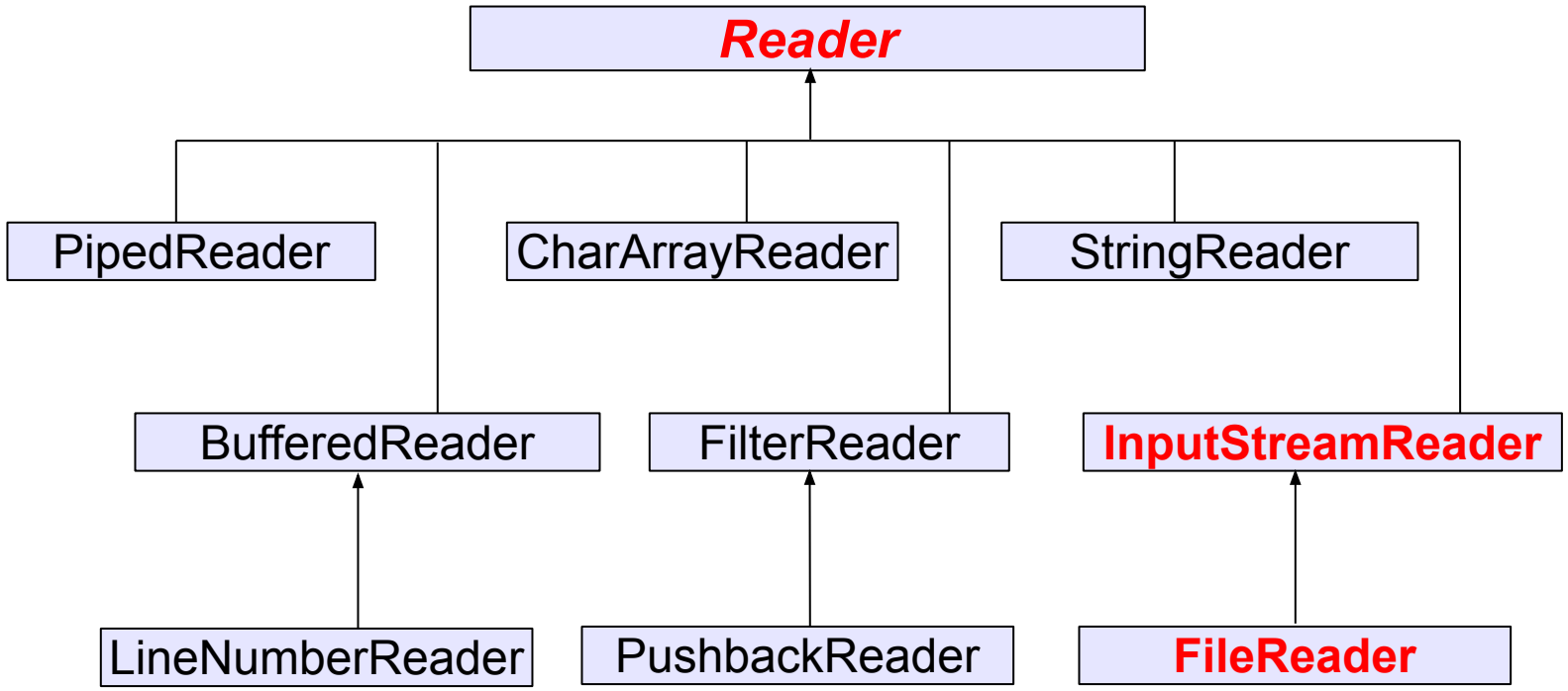
Character-Oriented `Writer` Classes

- The following is the ***character-oriented output stream*** class hierarchy:



Character-Oriented `Reader` Classes

- The following is the ***character-oriented input stream*** class hierarchy:



softserve

`FileReader` and `FileWriter` Classes

```
String fileName = "data.txt";
String data = "Hello Java I/O! Streams";

try (FileWriter fileWriter = new FileWriter(fileName /*, true*/ ))
{

    fileWriter.write(data);

} catch (IOException e) {
    e.printStackTrace();
}

try (FileReader fileReader = new FileReader(fileName)) {

    int c = 0;
    while((c = fileReader.read()) != -1) {
        System.out.print( (char)c );
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

softserve

`BufferedWriter` and `BufferedReader` Classes

- The **`BufferedReader`** and **`BufferedWriter`** classes use an ***internal buffer*** to store data while reading and writing, respectively.
  - The **`BufferedReader`** class provides a new method **`readLine()`**, which reads a line and returns a **`String`** (without the line delimiter).
  - The **`BufferedWriter`** class provides a new method **`newLine()`**, which write a separator to the buffered writer stream.

BufferedWriter and BufferedReader Classes

```
String fileName = "data.txt";
String data = "Hello Java I/O Streams!";

try (BufferedWriter bufferedWriter = new BufferedWriter(
        new FileWriter(fileName , true))) {

    bufferedWriter.write(data);
    bufferedWriter.newLine();
    bufferedWriter.flush();

} catch (IOException e) {
    e.printStackTrace();
}
```

softserve

BufferedWriter and BufferedReader Classes

```
try (BufferedReader bufferedReader = new BufferedReader(
        new FileReader(fileName))) {

    String line = null;
    while((line = bufferedReader.readLine()) != null) {
        System.out.print(line);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

```
Hello Java I/O Streams!
Hello Java I/O Streams!
Hello Java I/O Streams!
Hello Java I/O Streams!
```

softserve

Useful Links

- Java Basic I/O Tutorial

  http://docs.oracle.com/javase/tutorial/essential/io

- Tutorialspoint Java.io package tutorial

  http://www.tutorialspoint.com/java/io

Thanks for attention!