10 Text and Binary I/O

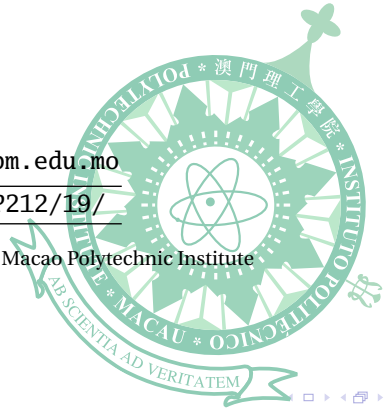
Instructor: Ke Wei（柯韋）

A319 Ext. 6452 wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

November 7, 2019



Outline

Strings

- A string is a sequence of characters.
- In many languages, strings are treated as arrays of characters, but in Java a string is an object.
- Java provides the *String* and *StringBuffer* classes for storing and processing strings.
- The *String* class is efficient for storing and processing strings, but strings created with the *String* class cannot be modified (immutable).
- The *StringBuffer* class enables you to create flexible strings that can be modified.

Constructing Strings

- You can create a *String* object from a string value or from an array of characters.
- The following statement creates a *String* object *message* for the string literal `"Welcome␣to␣Java"`:

```
String message = new String("Welcome␣to␣Java");
```

- Java treats a string literal as a *String object*. So the following statement is valid:

```
String message = "Welcome␣to␣Java";
```

- The following statements create the string `"Good␣Day"` from an array of characters.

```
char[] charArray = {'G','o','o','d','␣','D','a','y'};
String message = new String(charArray);
```

Interned Strings

- Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a *unique* instance for string literals with the same character sequence. Such an instance is called *interned*.
- For example,

```
String a = "Welcome to Java";
String b = new String("Welcome to Java");
String c = b.intern();
String d = "Welcome to Java";
System.out.println("a == b is " + (a == b)); // false
System.out.println("a == c is " + (a == c)); // true
System.out.println("a == d is " + (a == d)); // true
```

- In the preceding statements, *a, c* and *d* refer to the same interned string `"Welcome to Java"`. However, *b* is a new *String* object having the same content.

The *StringBuffer* Class

- *StringBuffer* is more flexible than *String*. You can add, insert, or append new contents into a string buffer.
- You can create an empty string buffer or a string buffer from a string.

```
StringBuffer strBuf = new StringBuffer();
StringBuffer strBufIni = new StringBuffer("Welcome");
```

- The *StringBuffer* class provides several overloaded methods to append and insert `boolean`, `char`, `char` array, `double`, `float`, `int`, `long`, and *String* into a string buffer.

```
strBuf.append("Hello ").append(909).append('!');
strBuf.insert(0, 3.14).insert(4, '?').insert(0, "What is ");
strBuf.append("OK, ").insert(strBuf.length(), 909);
```

More Operations on String Buffers

- You can also delete characters from a string in the buffer, reverse the string, replace characters, or set a new character in a string buffer.
- For example, suppose *strBuf* contains `"Welcome␣to␣Java"`,

strBuf.*delete*(`8, 11`)	changes the buffer to `"Welcome␣Java"`
strBuf.*deleteCharAt*(`8`)	changes the buffer to `"Welcome␣o␣Java"`
strBuf.*reverse*()	changes the buffer to `"avaJ␣ot␣emocleW"`
strBuf.*replace*(`11, 15, "HTML"`)	changes the buffer to `"Welcome␣to␣HTML"`
strBuf.*setCharAt*(`0, 'w'`)	sets the buffer to `"welcome␣to␣Java"`

The *File* Class

- Every file is placed in a directory in the file system.
- An absolute file name contains a file name with its complete path and drive letter on Windows.
- On Unix, the absolute file name may contain no drive letter.
- Absolute file names are machine-dependent.
- The *File* class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.
- The *File* class is a wrapper class for the file name and its directory path.
- The *File* class contains the methods for obtaining file properties and for renaming and deleting files.
- The *File* class does not contain the methods for reading and writing file contents.
- Constructing a *File* instance does not create a file on the machine.

Using the *File* Class

```
1  public static void main(String[] args) {
2      File file = new File("image/us.gif");
3      System.out.println("Does it exist? "+file.exists());
4      System.out.println("Can it be read? "+file.canRead());
5      System.out.println("Can it be written? "+file.canWrite());
6      System.out.println("Is it a directory? "+file.isDirectory());
7      System.out.println("Is it a file? "+file.isFile());
8      System.out.println("Is it absolute? "+file.isAbsolute());
9      System.out.println("Is it hidden? "+file.isHidden());
10     System.out.println("Absolute path is " + file.getAbsolutePath());
11     System.out.println("Last modified on " + new Date(file.lastModified()));
12 }
```

Writing Data Using *PrintWriter*

- The *PrintWriter* class can be used to write data to a text file.
- First, you have to create a PrintWriter object for a text file as follows:

```
PrintWriter output = new PrintWriter(file);
```

- Invoking the constructor will create a new file if the file does not exist. If the file already exists, the current content in the file will be discarded.
- Then, you can invoke the *print*, *println*, and *printf* methods on the *PrintWriter* object to write data to a file.

```
output.print("Hello ");
output.println("Java!");
```

- Finally, the *close*() method must be used to close the file. If this method is not invoked, the data may not be saved properly in the file.

Reading Data Using *Scanner*

- A *Scanner* breaks its input into tokens delimited by whitespace characters.
- To read from the keyboard, you create a *Scanner* for *System.in*, as follows:

```
Scanner input = new Scanner(System.in);
```

- To read from a file, create a *Scanner* for a file, as follows:

```
Scanner input = new Scanner(file);
```

- Invoking the constructor may throw an I/O exception, which is a checked exception.
- Then, you can invoke the *next* and *nextLine* methods on the *Scanner* object to read tokens and lines from a file.

```
String word = input.next();
String line = input.nextLine();
```

- The *close*() method releases the resources occupied by the file.

Reading Characters Using *FileReader*

- A *FileReader* reads individual characters without grouping them into tokens.
- To read characters from a file, create a *FileReader* for a file, as follows:

```
FileReader reader = new FileReader(file);
```

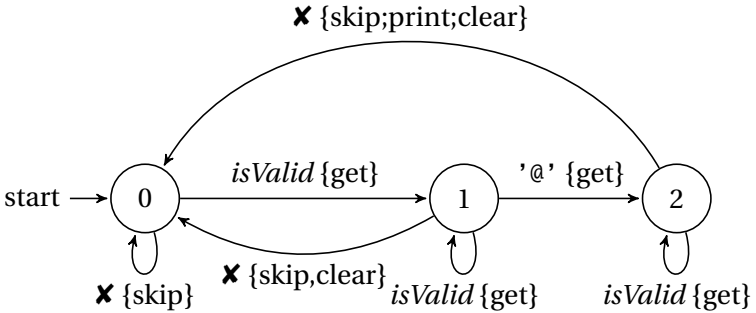
- The *FileReader* object can return the next character only when the *ready*() method returns true.
- Then, you can invoke the *read*() method on the *FileReader* object to read a character code. To get the character, you need to cast the code to a `char`, as follows:

```
while ( reader.ready() )
    System.out.print((char)reader.read());
```

- It is a good practice to call the *close*() method to release the resources occupied by the file.

Practice: Extracting Email Addresses

Given a text file, we need to find the email addresses in the file using an automaton, and write the found email addresses to a new text file.



(✘ — otherwise)

Input and Output Streams

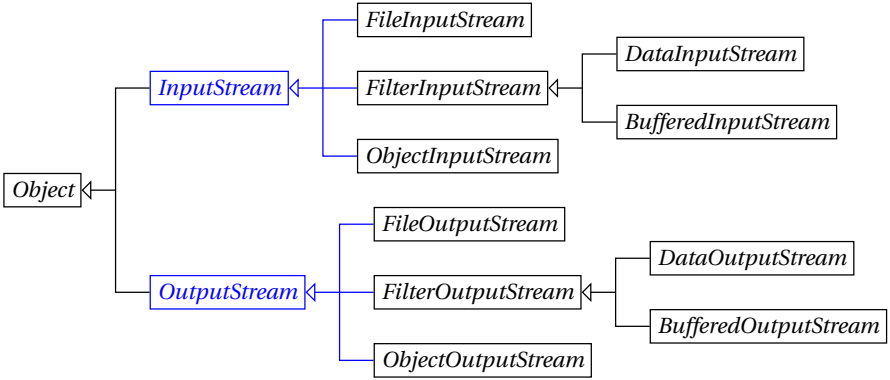
- A *File* object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file.
- In order to perform I/O, you need to create objects using appropriate Java I/O classes.
- An input class contains the methods to read data, and an output class contains the methods to write data.
  - *PrintWriter* is an example of an output class, and *Scanner* is an example of an input class.
- An input object is also called an input stream, and an output object is also called an output stream.

Text I/O vs. Binary I/O

- Computers do not differentiate binary files and text files. All files are stored in binary format, and thus all files are essentially binary files.
- Text I/O is built upon binary I/O to provide a level of abstraction for character *encoding* and *decoding*.
- The JVM converts a Unicode to a file-specific encoding when writing a character and converts a file-specific encoding to a Unicode when reading a character.
- Binary I/O does not require conversions. If you write a numeric value to a file using binary I/O, the exact value in the memory is copied into the file. When you read a byte using binary I/O, one byte value is read from the input.
- You should use text input to read a file created by a text editor or a text output program, and use binary input to read a file created by a Java binary output program.

Binary I/O Classes

InputStream is the root for binary input classes, and *OutputStream* is the root for binary output classes.



The *InputStream* Abstract Class

- `abstract int` *read*`()` — reads the next byte of data from `this` input stream.
- `int` *read*`(byte[]` *b*`)` — reads some number of bytes from `this` input stream and stores them into the buffer array *b*.
- `int` *read*`(byte[]` *b*`, int` *off*`, int` *len*`)` — reads up to *len* bytes of data from `this` input stream into the segment at offset *ofs* of the buffer array *b*.
- `long` *skip*`(long` *n*`)` — skips over and discards *n* bytes of data from `this` input stream.
- `int` *available*`()` — returns an estimate of the number of bytes that can be read (or skipped over) from `this` input stream without blocking by the next invocation of a method for `this` input stream.
- `void` *close*`()` — closes `this` input stream and releases any system resources associated with the stream.
- `void` *mark*`(int` *readlimit*`)` — marks the current position in `this` input stream.
- `boolean` *markSupported*`()` — tests if `this` input stream supports the mark and reset methods.
- `void` *reset*`()` — repositions `this` stream to the position at the time the mark method was last called on `this` input stream.

The *OutputStream* Abstract Class

- `abstract void` *write*`(int` *b*`)` — writes the specified byte to `this` output stream.
- `void` *write*`(byte[]` *b*`)` — writes *b*.*length* bytes from the specified byte array to `this` output stream.
- `void` *write*`(byte[]` *b*`, int` *off*`, int` *len*`)` — writes *len* bytes from the specified byte array starting at offset *off* to `this` output stream.
- `void` *close*`()` — closes `this` output stream and releases any system resources associated with `this` stream.
- `void` *flush*`()` — flushes `this` output stream and forces any buffered output bytes to be written out.

FileInputStream and *FileOutputStream*

- *FileInputStream*/*FileOutputStream* is for reading/writing bytes from/to files.
- To construct a *FileInputStream*, use the following constructors
  - *FileInputStream*(*File file*)
  - *FileInputStream*(*String name*)

  A *FileNotFoundException* will occur if you attempt to create a *FileInputStream* with a nonexistent file.
- To construct a FileOutputStream, use the following constructors
  - *FileOutputStream*(*File file*)
    FileOutputStream(*File file*, `boolean` *append*)
  - *FileOutputStream*(*String name*)
    FileOutputStream(*String name*, `boolean` *append*)

  If the file does not exist, a new file will be created. If the file already exists, the *append* parameter specifies whether to retain the current content and append new data into the file, or to delete the current content of the file.

Example: Writing Bytes to a File and Reading Them Back

```
1  import java.io.*;
2  public class TestFileStream {
3      public static void main(String[] args) throws IOException {
4          try ( FileOutputStream output = new FileOutputStream("temp.dat") ) {
5              for (int i = 1; i <= 10; i++)
6                  output.write(i);
7          }
8
9          try ( FileInputStream input = new FileInputStream("temp.dat") ) {
10             int value;
11             while ( (value = input.read()) != -1 )
12                 System.out.print(value + " ");
13         }
14     }
15 }
```

DataInputStream and *DataOutputStream*

- Filter streams are streams that filter bytes for some purpose.
- *DataInputStream* reads bytes from the stream and converts them into appropriate primitive type values or strings. It extends *FilterInputStream* and implements the *DataInput* interface.
- *DataOutputStream* converts primitive type values or strings into bytes and outputs the bytes to the stream. It extends *FilterOutputStream* and implements the *DataOutput* interface.
- Data streams are used as wrappers on existing input, and output streams to filter data in the original stream. They are created using the following constructors,
  - *DataInputStream*(*InputStream instream*)
  - *DataOutputStream*(*OutputStream outstream*)
- The representation of a primitive type value in memory is the same as in a data stream.

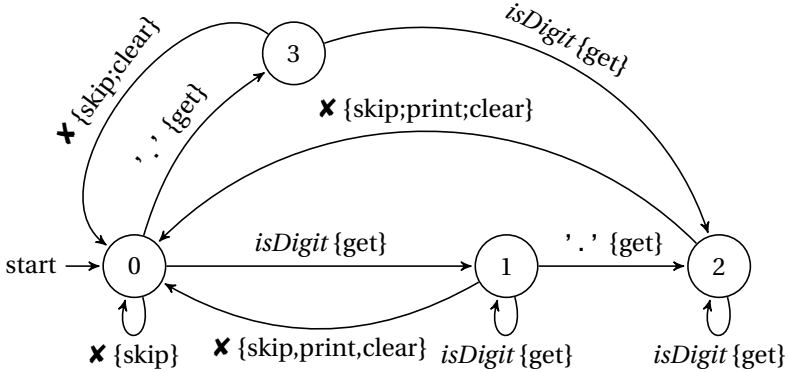
Example: Implementing an *InputStream*

```
1  public static void main(String[] args) throws IOException {
2      DataInputStream input = new DataInputStream(new InputStream() {
3          private byte[] a = {0,1,0,0,1,2,0,0,0,0,0,1,0,4,9,9};
4          private int i = 0;
5          @Override public int read() throws IOException {
6              if ( i < a.length ) return a[i++];
7              else return -1;
8          }
9      });
10
11     System.out.println(input.readShort());
12     System.out.println(input.readInt());
13     System.out.println(input.readLong());
14     System.out.println(input.readDouble());
15 }
```

```
1
258
65540
Exception in thread "main"
  java.io.EOFException
```

Homework: Extracting Decimal Numbers

Given a text file `input.txt`, write a program `ExtractDec.java` using the following automaton to find and extract the decimal numbers in the file, and print the extracted numbers to a new text file `output.txt`.



Homework: Extracting Decimal Numbers (2)

- Submit your code and data file, including `ExtractDec.java` and `output.txt` all in a zip file, within a week.
- Consider modifying the automaton to preserve the line structure of the original file.
- You can download the example program and the `input.txt` from the teacher's website.