COMP6700/2140 Input/Output

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

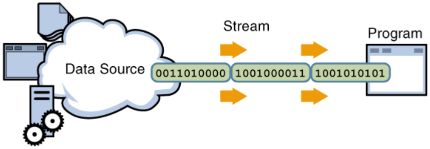
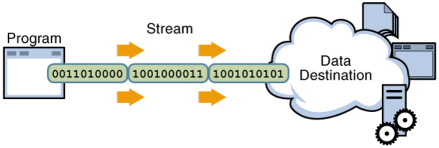
17 March 2017

I/O streams

From Java Tutorial

An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

Streams support many different kinds of data, including simple bytes, primitive data types, localised characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways. No matter how they work internally, all streams present the same simple model to programs that use them. A stream is a sequence of data.

Input	**Output**
A program uses an input stream to to read data from a source, one item at a time	A program uses an output stream to to write data to a destination, one item at a time
 Data Source Stream Program 0011010000 1001000011 1001010101	 Program Stream Data Destination 0011010000 1001000011 1001010101

java.io package

java.io defines I/O in terms of *streams*. Streams are ordered sequences of data that have a source (input streams) or destination (output streams). The I/O classes act as front ends to the specific details of OS, providing access to the system resources through files, peripheral devices (keyboard, display screen) and other means. Operations which can be carried out over the streams are provided by in interfaces and abstract classes. Concrete classes (*eg Filters*) may have additional methods.

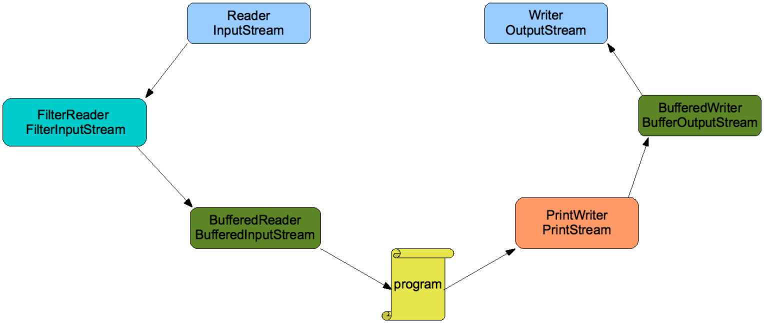
- *Character Streams* (16 bit, for text) and
  - *Readers* for input
  - *Writers* for output
- *Byte Streams* (8 bit, for data, e.g. images)
  - *Input Streams*
  - *Output Streams*

Stream are objects of corresponding I/O classes. Depending on the processing task, methods of the stream class are called, or the object is used as a parameter for a constructor of another stream, e.g. for filtering or buffering the original stream (see two examples, ByteCounter.java and UppercaseConverter.java):

```
// wrapping std input as a character stream
InputStreamReader cin = new InputStreamReader(System.in);
```

Input *to* and output *from* a program

Learning how to `read()` and `write()`



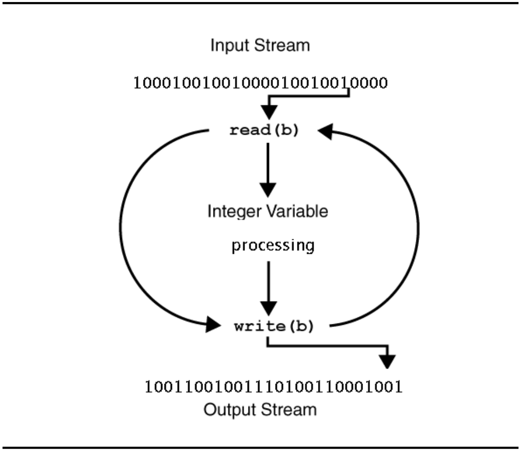
I/O *Stream* Classes

- *InputStream/OutputStream, Reader/Writer* — abstract classes on the top of corresponding hierarchies. Subclasses must always provide a method that returns the next byte of input/output — `read()/write()` (below we shall omit mentioning the byte stream classes).
- *FileReader/FileWriter* — for handling character files.
- *BufferedReader/BufferedWriter* — adds the ability to buffer the input (e.g. `readLine()` instead of a single character) and flush the output. A stream with additional capabilities (`br`) can be created (*Decorator* design pattern) from a more basic one (`fr`) as follows

```
FileReader fr = new FileReader(args[0]); // args[0] is a file name
BufferedReader br = new BufferedReader(fr);
//a shorter alternative
BufferedReader br1 = new BufferedReader(new FileReader("foo.in"));
```

- *FilterReader/FilterWriter* — abstract classes, which allow to manipulate (delete, replace) the stream character. These are examples of abstract classes with all methods implemented. Subclasses should override some of these methods and may also provide additional methods and fields.
- *PrintWriter* — a subclass of *Writer* with a number of overloaded `print()` and `printf()` methods that make it easy to write the values of primitive types and objects to a stream, in a human-readable text format.
- *Zip/Jar* input/output streams (in packages `java.util.zip` and `java.util.jar`) — stream filters for reading/writing files in the ZIP/JAR file formats. Includes support for both compressed and uncompressed data (no need to use *System* to extract or archive, neat!).

Read-in, process, write-out



Methods of *Stream* classes

- `read()` — reads a character from the stream; returns *int*; -1 means end of stream
- `write()` — writes a character/string to the stream
- `readLine()` (in *BufferedReader*) — reads a line
- `newLine()` (in *BufferedWriter*) — writes a line separator
- `flush()` (in *BufferedWriter*) — clears the buffer by writing it to the destination stream
- `print()`, `println()` (in *PrintWriter*) — converts a value of data (primitives and objects into a printable form and sends it to the stream
- `printf()` (in *PrintWriter*) — writes a formatted string to this writer using the specified format string and arguments; returns reference to `this` instance of *PrintWriter*
- `skip(long n)` (in *BufferedReader*) — skips *n* characters
- `reset()` — resets the stream (repositions it to point marked by `mark()`, or to the beginning)
- `close()` (in *BufferedWriter*) — flushes the buffer and closes the stream (when writing to a file, it is safer to `close()` and save the content)

Formatted Input Stream

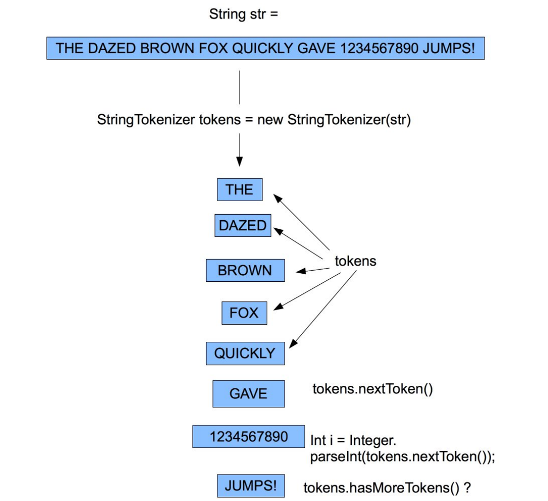
Problem: Process a formatted input — read an input text stream, organised line-by-line as a series of *fields* by breaking the lines and using each field accordingly.

The *StringTokenizer* class from `java.util` package is useful for processing formatted input. It breaks the input (when reading from a file line-by-line) into substrings which can be processed according to specified format. E.g. data from a file with astronomical records (fields) on *every line* including:

- `starName`
- `catNum`
- `class`
- `coor1,coor2,coor3`
- `luminosity`

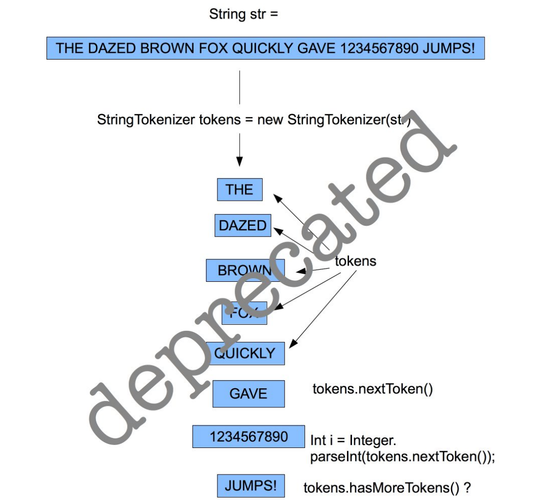
are read, parsed and then star objects created and added to a catalog.

String Tokenizer (old hat)



String str =
THE DAZED BROWN FOX QUICKLY GAVE 1234567890 JUMPS!
StringTokenizer tokens = new StringTokenizer(str)
THE
DAZED
BROWN
FOX
QUICKLY
GAVE
1234567890
JUMPS!
tokens
tokens.nextToken()
Int i = Integer.
parseInt(tokens.nextToken());
tokens.hasMoreTokens() ?

String Tokenizer (deprecated)



Alternative to Tokenizer

Forget `StringTokenizer` — it is deprecated!

StringTokenizer is a legacy class that is kept for backward compatibility, but its use is discouraged in new code. A better way to exercise the same functionality by using the `split()` method of *String*.

```
// "\\s" is a regex for any number of WS characters
String[] result = "this is a test".split("\\s");
for (String str: result)
    System.out.println(str + " has " + str.length());
output:
this has 4
is has 2
a has 1
test has 4
```

Note: apart from *StringTokenizer*, `java.io` package has ***Stream**Tokenizer*, which can recognise identifiers, numbers, quoted strings, and various comment styles. It is specially designed to process the text code for languages like Java and C. Unlike `java.util.StringTokenizer`, `java.io.StreamTokenizer` is **not deprecated**, it can be still quite useful.

Using String.split()

```
ArrayList<Star> stars; // a star catalog
BufferedReader input = new BufferedReader(new FileReader(catalog.txt));

String starName;
int catalogNumber;
char starClass;
Coordinates coor;
double lum;
String line = input.readLine();
String[] tokens;
while (line != null) {
    tokens = line.split("\\s"); // assume that number of tokens is
    starName = tokens[0];       // the same for every line
    catalogNumber = Integer.parseInt(tokens[1]);
    starClass = tokens[2].charAt(0);
    coor = new Coordinates(tokens[3]);
    lum = Double.parseDouble(tokens[4]);
    star = new Star(<all the collected values>);
    stars.add(star);
    line = input.readLine();
}
```

Scanner : the Great Simplifier

Scanner class combines the facilities of *InputStreams*, *StringTokenizer* and *Regex* classes to break down formatted input into tokens and translating individual tokens according to their data type. ***Scanner* is not a stream** (more like a *Tokenizer*), but it must be closed to indicate that it's finished with the underlying stream. The stream breaking is done in accordance with *delimiter pattern* (default is whitespaces, see `Character.isWhitespace(char c)`). The following example is taken directly from API documentation:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.print(s.nextInt() + " ");
System.out.print(s.nextInt() + " ");
System.out.print(s.next() + " ");
System.out.println(s.next());
s.close();
```

with the output (as you might expect)

```
1 2 red blue
```

Scanner allows to process input in accordance with specified *Locale*. The *Scanner* interface is quite big, but even the use of a small part of it can simplify your code substantially. Read API.

Regular Expression for Scanner

When a *Scanner* object scanner is created, the call to scanner.next() will return the next token. The default taken is a *word* (something which has whitespace in the front and at the end), but this can be changed.

- the example on the previous slide changes the separator between tokens.
- `scanner.useDelimiter("(?m:^$)");` – tokens are now paragraphs (separated by empty lines).
- `scanner.useDelimiter("\\A");` – read the whole file (less useful now in view of NIO, se below).
- **more examples**

Scanner is *AutoCloseable*

If an object implements `java.lang.AutoCloseable` — this means that when such an object is created inside *try-with-resources* block, it will destroy itself automatically when the block is exited (more discussion of try-with-resources is in P6 “Exceptions”). This is a simple example (courtesy of Jay Sridar blog on DZone) of how one can count words is a file:

```
try (Scanner scanner = new Scanner(new File(filename));) {
    int nword = 0;
    while (scanner.hasNext()) {
       String sent = scanner.next();
       nword++;
       System.out.printf("%3d %s%n", nword, sent);
    }
}
```

File Class and Directory Browsing

Doing the OS work from within a Java program

An application can interact with the underlying operating system. Often such interaction involves elements of the file system — files and directories. An application may require reading, writing and executing files, finding files, establishing and changing file attributes, deleting files *etc*. As usual, Java must do this in the OS-neutral way, without using OS-specific commands and file-system properties. This type of problems are dealt with with help of *File* class and a set of global configuration values called *properties* (which include *system properties*).

A *File* object is virtual proxy for an underlying OS file; it allows to find out everything about a file and perform a number of operations on it:

- name (`getName()`) and path (`getPath()`)
- mode (`canRead()`, `canWrite()`, `canExecute()`)
- status (`exists()`, `isDirectory()`, `isHidden()` *etc*)
- size (`length()`)
- create, rename and delete (`createNewFile()`, `renameTo()`, `delete()`)
- list directory content (`list()`, `listFiles()`)

and some others. As an example, this is how we can list a directory content for files which end with the chosen suffix (from examples DirectoryLister.java).

Example: Directory Listing from Within Java

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

class DirectoryLister {
    public static void main(String[] args) {
        String suffix = args.length < 1 ? "" : args[0]; // no "final"!
        //run-time directory name (different on different OS platforms)
        String cwdName = System.getProperty("user.dir");
        File cwd = new File(cwdName);  // cwd is File object standing for directory
        if (cwd.isDirectory()) {
            System.out.println("The directory contains:");
            for (File file: cwd.listFiles((f,n) -> n.endsWith(suffix)))
                System.out.printf("%s: %d\n", file.getName(), file.length());
        }
    }
}
```

Aside note: this is how it's done with Unix's command-line Smiley:

```
abx% ls -l *.suffix | awk '{ print $9": " $5 }'
```

Java's New I/O

Since Java 7, API offers substantial improvements in dealing with I/O (called `NIO.2`):

The class *File* from `java.io` allows full management of the file system — navigate the file hierarchy, establish file attributes, delete and rename files, link and unlink and so on.

But it works inconsistently across platforms e.g. handling of symbolic links; it does not always throw exceptions when expected, and has problems with synchronization (the file tree could change in the course of program execution, which would cause problems with navigating it via the absolute path).

The new class `java.nio.file.Path` addresses these problems. `NIO.2` also provides support for manipulating hard and soft links, controlling file attributes, and file change notification (when the application needs to detect and react to events of file system modifications) — instead of using polling mechanism with the old API, it is now possible to use a far simpler event-driven approach.

File Operation Support in NIO.2

Foundation classes / interfaces of the `java.nio.file` package include:

- `Path` — replaces `java.io.File`, includes methods to obtain information about the path, to access elements of the path, to convert the path to other forms, or to extract portions of a path. There are also methods for matching the path string and others.
- `Paths` — utility class with methods to return a path
- `FileSystem` — factory class that for objects in the filesystem
- `FileSystems` — utility with methods to access local or remote filesystems
- `WatchService` — utility class to detect file system changes through event notification; this allows *event-driven programming* style (which results in significant simplification)
- `Files` — utility class to create, rename, copy and delete files, changing their attributes etc (provides better support for *atomic* operations then `java.io.File`)

`Files` class supports navigating a full directory tree. It allows a program to easily search for files in a directory tree (including all nested directories) and perform operations on them as required e.g. delete those matching a pattern, copy, etc. The "magic" method is:

```
Files.walkFileTree(Path startingDir, FileVisitor<? super Path> visitor);
```

where the `FileVisitor` interface allows to program the perform particular operations on every traversed file or directory.

NIO.2 Examples

`NIO.2` offers simpler ways to work with files e.g. read / write, move from one location to another, copy, or delete.

```
List<String> lines = Files.
   readAllLines(Paths.get("file_to_read_from.txt"), StandardCharsets.ISO_8859_1);
for (String line : lines) {
   <if line matches a pattern, process and/or replace it>
}
Files.write(Paths.get("tmp.txt"), lines, Charset.forName("UTF-8"));
Files.copy(Paths.get("tmp.txt"), Paths.get("installed_packages.txt"),
         StandardCopyOption.REPLACE_EXISTING);
```

See example FindAndReplace.java

Some other useful `java.nio.file.Files` class methods:

- `Files.deleteIfExists(Paths.get("foo.txt"))` — returns true if deleted
- `Files.delete(Paths.get("bar.txt"))` — deletes a file or throws exception

Further Reading

- Hortsmann *Core Java for the Impatient*, Ch. 9.1, 9.2
- Oracle *The Java Tutorials*: Basic I/O