Java Input/Output

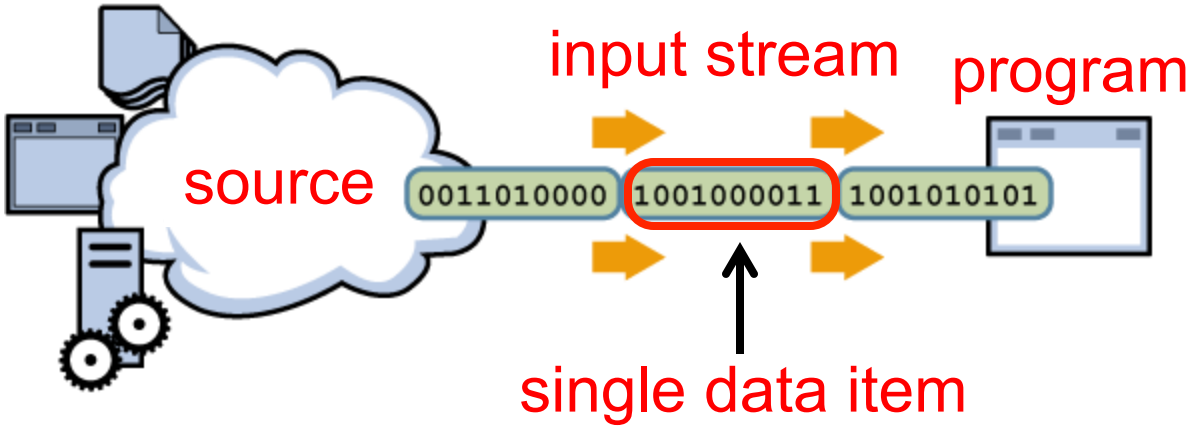
Overview

- The ***Java I/O (Input/Output)*** package `java.io` contains a group of interfaces and classes similar to the OSU CSE components' `SimpleReader` and `SimpleWriter` component families
  - Except that `java.io` is far more general, configurable, and powerful (and messy)
  - Hence, the names `SimpleReader` and `SimpleWriter`

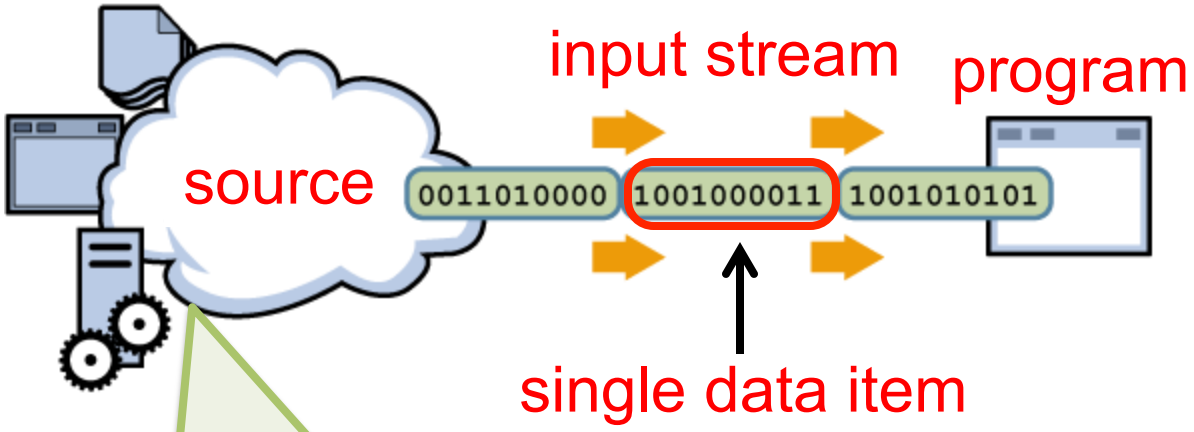
I/O Streams

- An ***input/output stream*** is a (conceptually not necessarily finite) series of data items
  - An ***input stream*** is a "flow" of data items from a ***source*** to a program
    - The program ***reads*** from the source (or from the stream)
  - An ***output stream*** is a "flow" of data items from a program to a ***destination***
    - The program ***writes*** to the destination (or to the stream)

Input Streams

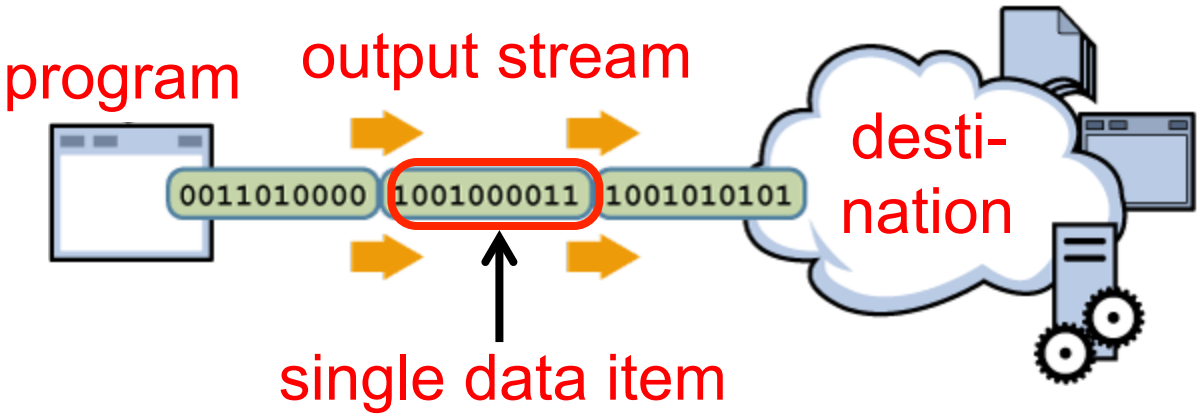


Input Streams

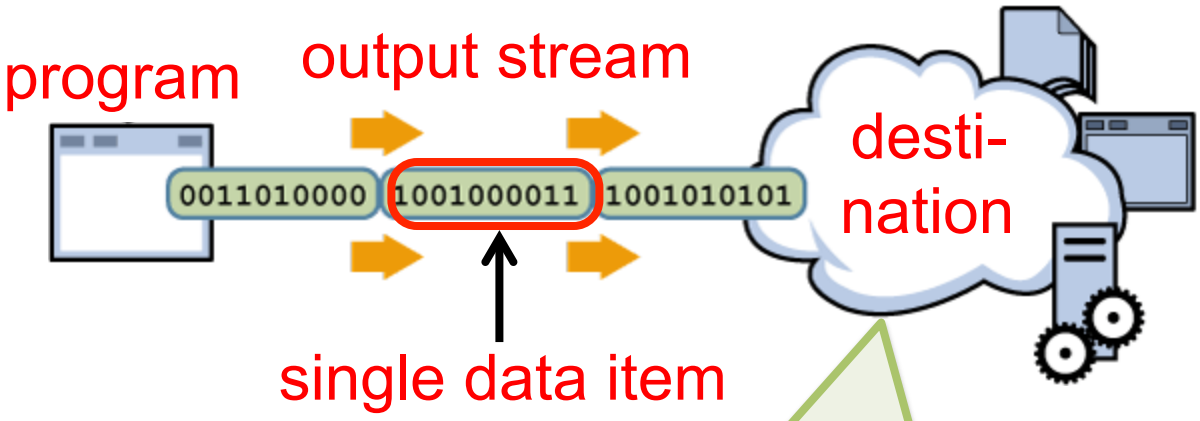


Source may be the keyboard, a file on disk, a physical device, another program, even an array or `String` in the same program.

Output Streams



Output Streams



Destination may be the console window, a file on disk, a physical device, another program, even an array or `String` in the same program.

Part I: Beginner's Guide

- This part is essentially a "how-to" guide for using `java.io` that assumes knowledge of the OSU CSE components' `SimpleReader` and `SimpleWriter` component families

Keyboard Input (`SimpleReader`)

- Here's some code in `main` to read input from the keyboard, using `SimpleReader`:

```
public static void main(String[] args) {
  SimpleReader input = new SimpleReader1L();
  String s = input.nextLine();
  ...
  input.close();
}
```

Advice (except for the simplest programs): to guard against the user entering "unexpected" input, read a line at a time into a `String` and then ***parse*** it to see whether it looks like expected.

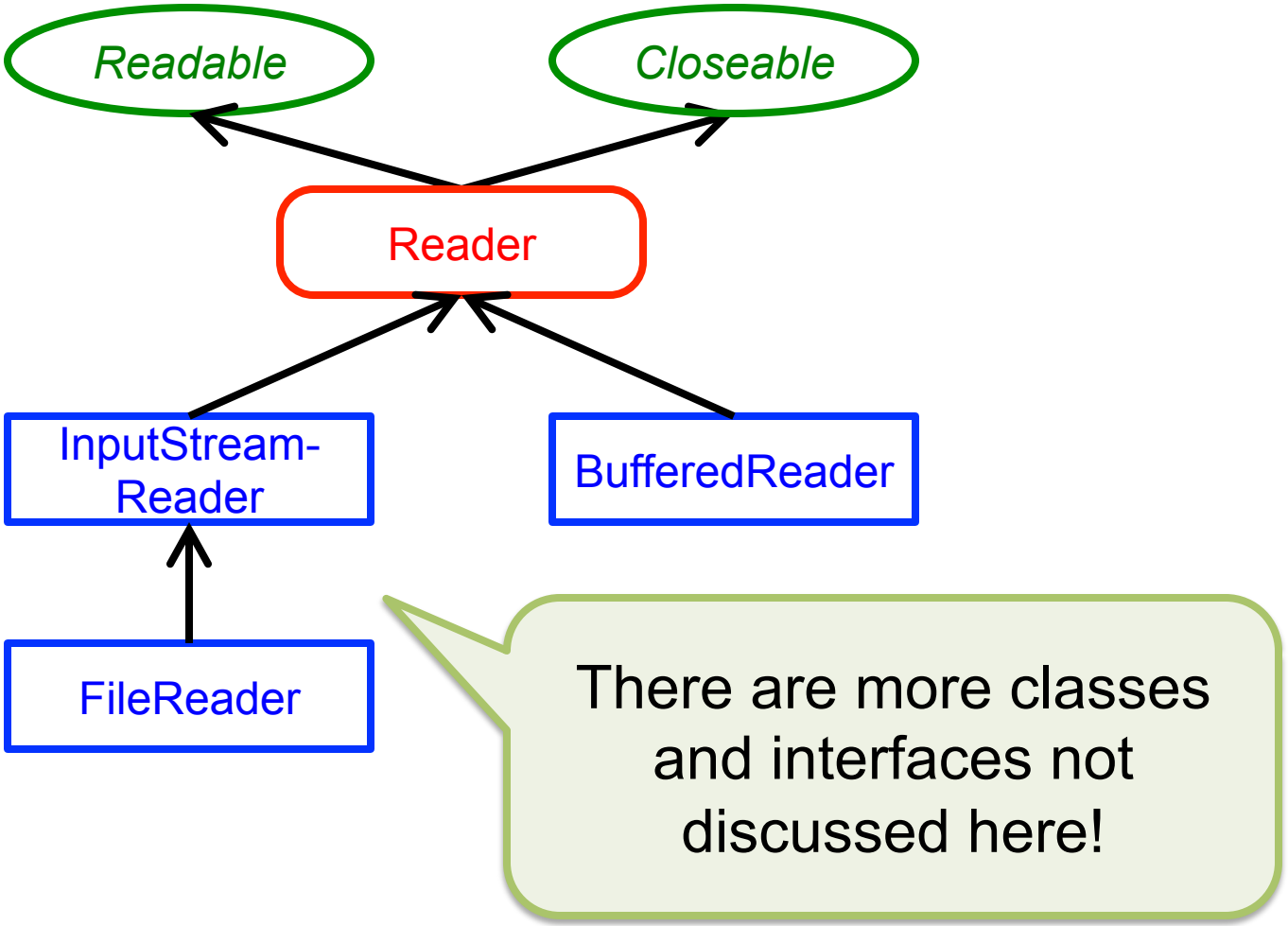
eReader)

ead input

leReader:

```
public static void ma      ing[] args) {
  SimpleReader input =  ew SimpleReader1L();
  String s = input.nextLine();
  ...
  input.close();
}
```

Overview of `java.io` Input



Keyboard Input (`java.io`)

- Here's some code in `main` to read input from the keyboard, using `java.io`:

```
public static void main(String[] args)
    throws IOException {
  BufferedReader input =
    new BufferedReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  ...
  input.close();
}
```

Some methods in `java.io` ***throw exceptions*** (discussed later) under certain circumstances, and you either need to ***catch*** them or let them propagate "up the call chain", like this.

a.io)

ead input

a.io:

```
public c void main(String[] args)
    throws IOException {
  BufferedReader input =
    new BufferedReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  ...
  input.close();
}
```

The variable `System.in` is a Java ***standard stream*** (discussed later), so you may use it without declaring it; but it is a ***byte stream*** (discussed later), so you want to ***wrap*** it ...

a.io)

ead input

d.io:

```
public static void mai          [] args)
    throws IOException {
  BufferedReader input =
    new BufferedReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  ...
  input.close();
}
```

... in a ***character stream*** (discussed later) like this ...

a.io)

ead input

.io:

```
publi        void main(String[] args)
    thr      Exception {
  Buffered   der input =
    new Buff redReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  ...
  input.close();
}
```

... and then you want to wrap that character stream in a ***buffered stream*** (discussed later), like this, so ...

a.io)

ead input

a.io:

```
public          void main(String[] args)
    thro        Exception {
  BufferedR     der input =
    new BufferedReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  ...
  input.close();
}
```

a.io)

ead input

a.io:

... you can read one line at a time into a `String`, like this.

```
public static void m          g[] args)
    throws IOExceptio
  BufferedReader inpu
    new BufferedReader
      new InputStreamRe der(System.in));
  String s = input.readLine();
  ...
  input.close();
}
```

Keybo

- Here's sor
from the k

```
public static void          tring[] args)
    throws IOExc        ion {
  BufferedReader input =
    new BufferedReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  ...
  input.close();
}
```

The ***declared*** types of variables when you use `java.io` are ***class types*** rather than ***interface types***.

Keyboa

- Here's sor
from the k

```
public static
    throws IOException {
  BufferedReader input =
    new BufferedReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  ...
  input.close();
}
```

This technique of slightly extending features or capabilities of an object by ***wrapping*** it inside another object is a popular object-oriented design pattern called the ***decorator*** pattern.

An Alternative (`java.util`)

- An attractive alternative to using `java.io.BufferedReader` is to use `java.util.Scanner`
  - Important difference: no `IOException`s can be raised, so you don't need to worry about catching or throwing them
  - Features for ***parsing*** input are powerful, e.g., the use of ***regular expressions*** to describe delimiters between data items

An Alternative (`java.util`)

- Here's some code in `main` to read input from the keyboard, using `Scanner`:

```
public static void main(String[] args) {
   Scanner input = new Scanner(System.in);
   String s = input.nextLine();
   ...
   input.close();
}
```

Now, `main` does not declare that it **`throws`** `IOException`; in this sense it is similar to using `SimpleReader`.

`util`)

ead input

from the keyboard, using `anner`:

```
public static void main(String[] args) {
  Scanner input = new Scanner(System.in);
  String s = input.nextLine();
  ...
  input.close();
}
```

Notice that initialization does not involve layers of wrapping of one object inside another; `Scanner` also has different constructors so it can be used with files.

util)

ead input

ner:

```
public static void mai   ing[] args) {
  Scanner input = new Scanner(System.in);
  String s = input.nextLine();
  ...
  input.close();
}
```

The method for reading a line into a `String` has a different name than for `BufferedReader`: it is `nextLine` (like `SimpleReader`).

util)

ead input

ner:

```
public static void ma      ing[] args) {
  Scanner input = new      nner(System.in);
  String s = input.nextLine();
  ...
  input.close();
}
```

End-of-Stream (`SimpleReader`)

```
public static void main(String[] args) {
  SimpleReader input = new SimpleReader1L();
  while (!input.atEOS()) {
    s = input.nextLine();
    ...
  }
  ...
  input.close();
}
```

End-of-Stream (`SimpleReader`)

```
public static void main(String[] args) {
  SimpleReader input = new SimpleReader1L();
  while (!input.atEOS()) {
    s = input.nextLine();
    ...
  }
  ...
  input.
}
```

`SimpleReader` has a method to report "at end-of-stream", and it can tell you this *before* you read "past the end of the input stream"; similarly, `Scanner` has method `hasNext`.

End-of-Stream (java.io)

```
public static void main(String[] args)
    throws IOException {
  BufferedReader input =
    new BufferedReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  while (s != null) {
    ...
    s = input.readLine();
  }
  ...
  input.close();
}
```

End-o

```
public stati
    throws I
  BufferedRe
    new BufferedRe
      new InputStre    Reader(System.in));
  String s = input.readLine();
  while (s != null) {
    ...
    s = input.readLine();
  }
  ...
  input.close();
}
```

`BufferedReader` has no method to detect when you cannot read any more input, so you can check it only *after* you try to read "past the end of the stream".

End-of-Stream (`java.io`)

```
public static void main(String[] args)
    throws IOException {
  BufferedReader input =
    new BufferedReader(
      new InputStreamReader(System.in));
  String s = input.readLine();
  while (s != null) {
    ...
    s = input.readLine();
  }
  ...
  input.close();
}
```

Before checking again, you must try to read another line.

File Input (`SimpleReader`)

- Here's some code in `main` to read input from a file, using `SimpleReader`:

```
public static void main(String[] args) {
  SimpleReader input =
    new SimpleReader1L("data/test.txt");
  String s = input.nextLine();
  ...
  input.close();
}
```

ader)

ead input

er:

```
public static void mai     g[] args) {
  SimpleReader input =
    new SimpleReader1L("data/test.txt");
  String s = input.nextLine();
  ...
  input.close();
}
```

`SimpleReader` has a constructor from a `String`, which is the name of the file you want to read from.

File Input (`java.io`)

- Here's some code in `main` to read input from a file, using `java.io`:

```
public static void main(String[] args)
    throws IOException {
  BufferedReader input =
    new BufferedReader(
      new FileReader("data/test.txt"));
  String s = input.readLine();
  ...
  input.close();
}
```

Now, you wrap a `BufferedReader` around a `FileReader` (which has a constructor from a `String`, which is the name of the file you want to read from).

```
public static void ma          g[] args)
    throws IOException {
  BufferedReader input =
    new BufferedReader(
      new FileReader("data/test.txt"));
  String s = input.readLine();
  ...
  input.close();
}
```

Independence From Source

- With `SimpleReader`, `BufferedReader`, and `Scanner` used as shown, the source is identified in a constructor call, and subsequent code to read data is ***independent*** of the source
  - This is a really handy feature in many software maintenance situations
  - But notice: methods differ between `BufferedReader` and the other two!

Console Output (`SimpleWriter`)

- Here's some code in `main` to write output to the console, using `SimpleWriter`:

```
public static void main(String[] args) {
  SimpleWriter output = new SimpleWriter1L();
  output.print("foo");
  output.println(" and bar");
  ...
  output.close();
}
```

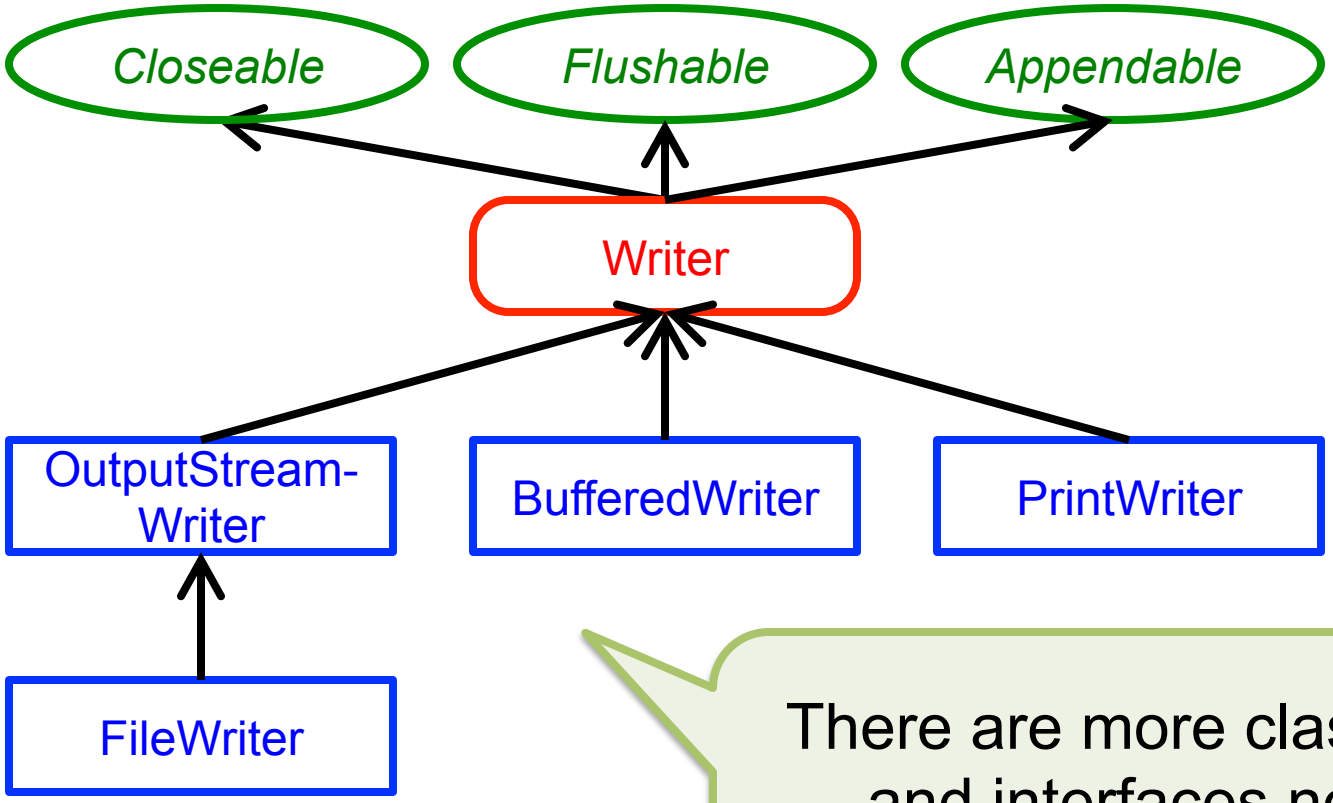
(...eWriter)

...vrite output
...Writer:

```
public ... void main(String[] args) {
  SimpleWr...er output = new SimpleWriter1L();
  output.print("foo");
  output.println(" and bar");
  ...
  output.close();
}
```

Both `print` and `println` are available, and the latter should be used to output the ***line separator*** string for the current "platform" (i.e., the system the Java program is running on).

Overview of `java.io` Output



Console Output (`java.io`)

- Here's some code in `main` to write output to the console, using `java.io`:

```
public static void main(String[] args) {
   ...
   System.out.print("foo");
   System.out.println(" and bar");
   ...
}
```

a.io)

vrite output

:

```
publi    ic void main(String[] args) {
  ...
  System.out.print("foo");
  System.out.println(" and bar");
  ...
}
```

`System.out` is another Java ***standard stream*** (discussed later), which you neither declare nor close; both `print` and `println` are available.

It is fine to use `System.out` for console output, but there are potential problems using (*unwrapped*) `System.in` for keyboard input.

a.io)

vrite output

:

```
publi    ic void main(String[] args) {
  ...
  System.out.print("foo");
  System.out.println(" and bar");
  ...
}
```

File Output (`SimpleWriter`)

- Here's some code in `main` to write output to a file, using `SimpleWriter`:

```
public static void main(String[] args) {
  SimpleWriter output =
    new SimpleWriter1L("data/test.txt");
  output.print("foo");
  output.println(" and bar");
  ...
  output.close();
}
```

riter)

rite output

:

```
public static void mai     g[] args) {
  SimpleWriter output =
    new SimpleWriter1L("data/test.txt");
  output.print("foo");
  output.println(" and bar");
  ...
  output.close();
}
```

`SimpleWriter` has a constructor from a `String`, which is the name of the file you want to write to.

File Output (`java.io`)

- Here's some code in `main` to write output to a file, using `java.io`:

```
public static void main(String[] args)
    throws IOException {
  PrintWriter output =
    new PrintWriter(
      new BufferedWriter (
        new FileWriter("data/test.txt")));
  output.print("foo");
  output.println(" and bar");
  ...
  output.close();
}
```

io)

...rite output

PrintWriter is needed for convenient output, and is added as yet another wrapper—to get print and println.

```
public static void m          g[] args)
    throws IOException
  PrintWriter output =
    new PrintWriter(
      new BufferedWriter (
        new FileWriter("data/test.txt")));
  output.print("foo");
  output.println(" and bar");
  ...
  output.close();
}
```

Independence From Destination

- With `SimpleWriter`, `System.out`, and `PrintWriter` used as shown, the destination is identified in a constructor call, and subsequent code to write data is ***independent*** of the destination
  - This is a really handy feature in many software maintenance situations
  - Unlike input, methods (`print` and `println`) for all three have same names and behaviors

Part II: Some Details

- There are way too many details about `java.io` to cover here; see the Javadoc and the *Java Tutorials* trail on "Basic I/O"
- A few details previously promised ...

IOException

- A general discussion of ***exceptions*** in Java is to come later still, but for now…
- A number of `java.io` constructors and methods might ***throw (raise)*** an `IOException`
  - Examples: files to be used as sources/ destinations may not exist, may not be readable and/or writeable by the user of the program, etc.

Try-Catch

- As an alternative to letting an exception propagate “up the call chain” to the client (as in the earlier examples), you may deal with an `IOException` by ***catching (handling)*** it, e.g.:
  - Report that there has been an I/O error and exit “gracefully”
  - Try to recover from it (which is usually much harder)

Example

- Here's the overall structure of an example `main` in a simple application that reads input from a file, using `java.io`:

```
public static void main(String[] args) {
   // Code to open file
   // Code to read from file
   // Code to close file
}
```

Example: Opening a File

```
public static void main(String[] args) {
  BufferedReader input;
  try {
    input = new BufferedReader(
              new FileReader("data/test.txt"));
  } catch (IOException e) {
    System.err.println("Error opening file");
    return;
  }
  // Code to read from file
  // Code to close file
}
```

Example: Opening a File

```
public static void main(String[] args) {
  BufferedReader input;
  try {
    input = new BufferedReader(
              new FileReader("data/t      xt"));
  } catch (IOException e) {
    System.err.println("Error openi          );
    return;
  }
  // Code to
  // Code to
}
```

Now, `main` does not declare that it **`throws`** `IOException` if it (*always*) ***catches*** the exception.

Example: Opening a File

```
public static void main(String[] args) {
  BufferedReader input;
  try {
    input = new B   feredReader(
                new    eReader("data/test.txt"));
  } catch (IOExcep      e) {
    System.err.pri      rror opening file");
    return;
  }
  // Code to
  // Code to
}
```

This variable must be declared (but not initialized) here, so it is in scope later where the code reads from the file—*if* the file is opened successfully.

Example: Opening a File

```
public static void main(String[] args) {
  BufferedReader input;
  try {
    input = new BufferedReader(
              new FileReader("data/test.txt"));
  } catch (IOException e) {
    System.err.println("Error opening file");
    return;
  }
  // Code to
  // Code to
}
```

The ***try block*** contains the code that might throw an `IOException` ...

Example: Opening a File

```
public static void main(String[] args) {
  BufferedReader input;
  try {
    input = new BufferedReader(
                new FileReader("data/test.txt"));
  } catch (IOException e) {
    System.err.println("Error opening file");
    return;
  }
  // Code to
  // Code to
}
```

... which this constructor might throw, e.g., if the file does not exist.

Example: Opening a File

```
public static void main(String[] args) {
  BufferedReader input;
  try {
    input = new BufferedReader(
                new FileReader("data/test.txt"));
  } catch (IOException e) {
    System.err.println("Error opening file");
    return;
  }
  // Code to
  // Code to
}
```

The ***catch block*** states the exception it handles, and is executed iff code in the try block throws that exception (which is called `e` in the catch block).

Example: Opening a File

```
public static void main(String[] args) {
  BufferedReader input;
  try {
    input = new BufferedReader(
                new FileReader("data/test.txt"));
  } catch (IOException e) {
    System.err.println("Error opening file");
    return;
  }
  // Code to
  // Code to
}
```

Here, the code prints an error message to `System.err`, another Java ***standard stream*** (discussed later), and "aborts" `main` by returning from it.

Exan

```
public stati
  BufferedRe
  try {
    input = new F        Reader(
                n        eReader("data/test.txt"));
  } catch (IOE        ion e) {
    System.e        rintln("Error opening file");
    return;
  }
  // Code to read from file
  // Code to close file
}
```

In either case (even after the catch block, if it did not **`return`** as it does here), execution proceeds with the next statement.

Example: Reading From a File

```
public static void main(String[] args) {
  // Code to open file
  try {
    String s = input.readLine();
    while (s != null) {
      ...
      s = input.readLine();
    }
  } catch (IOException e) {
    System.err.println("Error reading from file");
  }
  // Code to close file
}
```

Example

We need this try-catch because the method `readLine` might throw an `IOException`.

```
public st
  // C   e to
  try {
    String s = input.readLine();
    while (s != null) {
      ...
      s = input.readLine();
    }
  } catch (IOException e) {
    System.err.println("Error reading from file");
  }
  // Code to close file
}
```

Example

```
public stati
  // Code to
  try {
    String s = inp          ine();
    while (s != n
      ...
      s = input.          ine();
    }
  } catch (IOEx          ption e) {
    System.err          rintln("Error reading from file");
  }
  // Code to close file
}
```

As before, even after the catch block (which in this case does not end with a **return**), execution proceeds with the next statement.

Example: Closing a File

```
public static void main(String[] args) {
  // Code to open file
  // Code to read from file
  try {
    input.close();
  } catch (IOException e) {
    System.err.println("Error closing file");
  }
}
```

Exar

We need this try-catch because even the method `close` might throw an `IOException`.

```
public stati
   // Code to
   // Code to      from file
   try {
     input.close();
   } catch (IOException e) {
     System.err.println("Error closing file");
   }
}
```

The Standard Streams

- The utility class `System` in `java.lang` declares three ***standard streams***:
  - `System.in`
  - `System.out`
  - `System.err`
- You do not declare, open, or close these streams; but you can always use them without worrying about exceptions

The Standard Streams

- The utility class `System` in `java.lang` declares thr*** ***standard streams***:
  - `System.in`
  - `System.`
  - `System.`
- You do no
  streams; b
  without worrying about exceptions

A ***utility class*** is a class that cannot be instantiated, i.e., there is no public constructor; it is not a ***type***, so you cannot create "an object of that type".

The Standard Streams

- The utility clas
  declares three
  - `System.in`
  - `System.out`
  - `System.err`
- You do not declare, open, or close these streams; but you can always use them without worrying about exceptions

`System.err` is intended for error messages; `System.out` is intended for normal output.

Byte and Character Streams

- Java has two categories of streams:
  - ***Byte streams*** are streams of 8-bit bytes
    - This is a kind of low-level I/O that you rarely need
  - ***Character streams*** are streams of Unicode characters
    - This is preferred for text I/O because it accounts for the "local" character set and supports ***internationalization*** with little additional effort
- **Best practice** is to use character streams with textual I/O

Buffered Streams

- ***Buffered streams*** minimize disk access for reading/writing files, and generally have performance advantages over unbuffered streams
- **Best practice** is to "wrap" input and output character streams to create buffered versions
  - This is done with `BufferedReader` and `BufferedWriter`, as seen earlier

Files and Paths

- The `File` class (an original part of Java) and the `Path` interface (new in Java 1.7) allow you to manipulate directories and files and the "paths" to them in the file system, e.g.:
  - Check file existence and permissions, create files and set permissions, delete files, etc.
- See the *Java Tutorials*: Basic I/O and the Java libraries' Javadoc for details

Resources

- *The Java Tutorials*: Basic I/O
  - http://docs.oracle.com/javase/tutorial/essential/io/index.html