File Input

Reading Files

- We have already seen that writing to files is (can be?) similar to using `System.out`.
- Not surprisingly, reading files is similar to using `System.in`.
  - … with a few extra wrinkles.
- Again, the classes needed to do this are in `java.io`.

The `Reader` Class

- The `Reader` class is used to read text.
  - i.e. any stream of characters
  - It's abstract: the parent of many subclasses.
  - The counterpart of `Writer`.
- Provides the `read()` method.
  - reads single characters or character arrays.
  - Reads as many characters as it can, up to the array size.

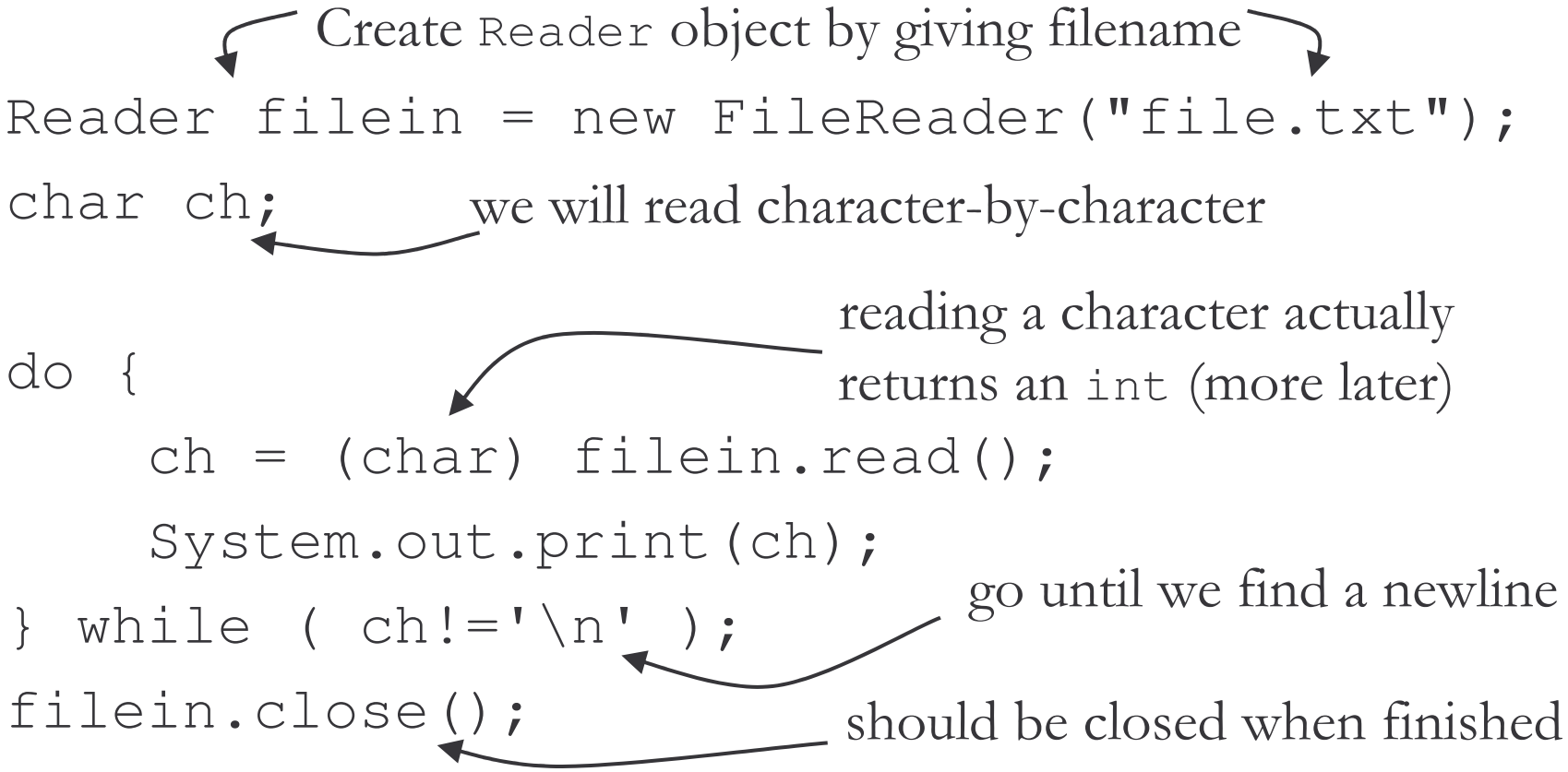
Reading Files

- The `FileReader` class is analogous to `FileWriter`.
  - a subclass of `Reader` that can work on a file.
  - constructor takes a filename as its argument.
- e.g.

```
Reader filein =
    new FileReader("somefile.txt");
```

Example

- This reads and prints the first line of the file.



End of File

- When reading from `System.in`, we just waited for the user to type more.
- When reading from a file, there might not be any more file to read.
  - Our programs will have to handle the possibility that we've reached the end of the file.
  - Different `Reader` methods have different ways of signaling the end of file.

End of File in a `Reader`

- `.read()`: returns an integer, not a character. Returns -1 if end of file. Can cast to a character otherwise.
- `.read(char[])`: puts characters into the array, returns the number of characters read; -1 if end of file.
  - So not all of the array will have meaningful contents, only characters 0 to return value.

Working with `Reader`

- Makes working with `Reader` tricky:

```
Reader filein = new FileReader("test.txt");
char[] buf = new char[10];
int numChars;
while ( true ) {
    numChars= filein.read(buf);
    if ( numChars == -1 ) { break; }
    System.out.print("("
        + new String(buf, 0, numChars) + ")");
}
```

read into a `char` array

read as much as possible

EOF when it returns -1:
exit loop

convert characters
0…numChars-1 to a `String`

Buffered Reading

- Many small reads are also inefficient.
- The operating system typically reads ahead in the file and stores the next parts in memory.
  - "disk cache"
- The `BufferedReader` class enhances this.
  - Read the next part of the file into memory and just returns it when we try to read more.
  - Speedup similar to `BufferedWriter`: a few times.

BufferedReader

- Wraps an existing `Reader` object:

```
Reader filein = new BufferedReader(
        new FileReader("file.txt") );
```

- Otherwise works exactly like any other `Reader` object.
  - … but faster.

Scanner

- We have already used a `Scanner` object to wrap the `System.in` stream:

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

- `System.in` is (almost) a `Reader` object.
  - We can treat any other `Reader` object the same way.

Example

- Read (and sum) integers in a file.

```
Reader filein = new BufferedReader(
          new FileReader("numbers.txt") );
Scanner scanfile = new Scanner(filein);
int total = 0;

while ( scanfile.hasNextInt() ) {
    total += scanfile.nextInt();
}
System.out.println(total);
```

Construct `Scanner` with a `Reader`

knows how to check for more input

… and convert appropriately

Incorrect Input

- When reading, the input might not be in the form you expect.
- If using a plain `Reader`, you have to process the characters yourself.
  - Whatever logic does this should carefully check input to make sure it's in the right form.
  - It should only be accepted if it's in the right form.
  - If not, the program should fail gracefully.

Bad Input and `Scanner`

- Calling `nextWhatever()` on a `Scanner` object throws an exception if it can't read the appropriate type.
- There are different reasons this could happen.
  - End of file (`NoSuchElementException`)
  - Data doesn't look like the corresponding type (`InputMismatchException`)
    - eg. "`abcd`" having called `nextInt()`

Bad Input and `Scanner`

- The `hasNextWhatever()` methods will tell you (without an exception) if `nextWhatever()` will succeed.
  - Could be false because of end of file or bad input.
  - Be careful that that's what you really want to check.
- e.g. previous example will stop as soon as there are any non-integers in the file.

Dealing with Bad Input

- For keyboard input, you can ask the user to re-enter the value.
- When reading a file that isn't possible.
  - You should always consider what the "right" thing to do is.
  - Fail completely? Read data up to that point? Skip to next line? Skip next character? Ask the user?
- Can be very difficult, depending on the data.

Example

- Read & sum integers, but handle errors.
  - On bad input, throw away rest of line and try again.
- We will also catch other checked exceptions.
  - `main()` will no longer have to have a `throws` clause.

Example, part 1

- Open file, catch exceptions, init variables:

```
Reader filein;
try {
    filein = new FileReader("numbers.txt");
} catch (FileNotFoundException e) {
    System.err.println("Couldn't open file.");
    return;  // exit main() & stop program
}
Scanner scanfile = new Scanner(filein);
int total = 0;
boolean done = false;
```

Example, part 2

```
while ( !done ) {
    try {
        total += scanfile.nextInt();
    } catch (InputMismatchException e) {
        // not an integer: eat rest of line
        scanfile.nextLine();
    } catch (NoSuchElementException e) {
        // end of file
        done = true;
    }
}
System.out.println(total);
```