Simple Text I/O

java.util.Scanner

- Java finally has a fairly simple way to read input
- First, you must create a `Scanner` object
  - To read from the keyboard (`System.in`), do:
    - `Scanner scanner = new Scanner(System.in);`
  - To read from a file, do:

    -
      ```
      File myFile = new File("myFileName.txt");
      Scanner scanner = new Scanner(myFile);
      ```
    - You have to be prepared to handle a `FileNotFound` exception
  - You can even “read” from a String:
    - `Scanner scanner = new Scanner(myString);`
    - This can be handy for parsing a string

Preparing to read

- You can test if there is something more to read:
  - scanner.hasNext()
- And you can test what kind of thing it is:
  - hasNextBoolean()
  - hasNextByte()
  - hasNextShort()
  - hasNextInt()
  - hasNextLong()
  - hasNextFloat()
  - hasNextDouble()
- hasNext() is used when reading from *files*
- When reading from the keyboard, hasNext() will always return true
  - This is because Java has no way of telling you aren't going to enter anything more
- These methods "peek" at the next thing, but they *do not* read it or go past it
- To skip over one token, call next() (you can ignore what it returns)
- To skip the rest of a line, call nextLine() (you can ignore what it returns)

Reading

- You can read the next token in as a String:
  - `String token = sc.next();`
- You can read the next token and have it automatically converted to a primitive value:
  - `boolean b = sc.nextBoolean();`
  - `byte by    = sc.nextByte();`
  - `short sh   = sc.nextShort();`
  - `int i      = sc.nextInt();`
  - `long l     = sc.nextLong();`
  - `float f    = sc.nextFloat();`
  - `double d  = sc.nextDouble();`
- You can read an entire line:
  - `String line = sc.nextLine();`
- A "token" is a sequence of printable characters delimited by whitespace
  - It's possible to use different delimiters; see the API
- If you ask to read in one kind of primitive, but get a different kind, Java throws an Exception
  - This is fine when reading from a file that has a required format
  - When reading from keyboard, you should peek ahead before you read a primitive, to see if you will get the right kind of thing from the user
- nextLine() returns everything remaining on the line (but discards the terminating newline character)

Recognizing newlines

- When reading from the keyboard, the Scanner doesn't read anything until the user presses Enter
  - However, the program just sees a sequence of tokens, and doesn't know what line a token comes from
  - For example, if you read a sequence of numbers, you can't tell whether they are on separate lines, or all on one line
- If it's important to know what line a token is on:
  - Use nextLine() to read in an entire line at a time
  - Create *another* scanner to scan this line (as a String)
  - Get your inputs from this second scanner
  - Use hasNext() to tell when you are at the end of the line
- **Important note:** *Users make mistakes!*
  - If you ask the user for a particular kind of value, don't assume that's the kind of value you will get--use the appropriate hasNextXXX() method
  - Once you get the value, test if it makes sense

Formatted output

- `System.out.println(Math.PI);`
  will print out
  `3.141592653589793`
  - If you want to print out this number as `3.1416`, or `3.14`, you need to *format* it
  - If you want to print out numbers in neat columns, you need to *format* them
- Prior to Java 1.5, you had to figure out how to do this yourself
  - Java 1.5 introduced the `Formatter` class to do formatting for you
  - In typical Java style, `Formatter` can do just about *anything—but* doesn't try to make the common things *easy*
- For the most part, we won't use the `Formatter` class directly, but will use `System.out.format(...)`

Formatted output

- Java 5 has a printf method, similar to that of C
- Each format code is % ***width code***
  - The ***width*** is the number of characters that are output (with blank fill)
    - By default, output is right-justified
    - A negative width means to left-justify the output
  - Some values for the ***code*** are s for strings, d for integers, f for floating point numbers, b for booleans
    - For floating point numbers, the ***width*** has the form ***total.right***, where ***total*** is the total width and ***right*** is the number of digits to the right of the decimal point
  - There are a huge number of options for formatting dates, which we won’t cover

Examples

-
  ```
  System.out.printf("Left justified:  |%-8s|\n", "abc");
  System.out.printf("Right justified: |%8s|\n", "abc");
  System.out.printf("Left justified:  |%-8d|\n", 25);
  System.out.printf("Right justified: |%8d|\n", 25);
  System.out.printf("Left justified:  |%-8.4f|\n", Math.PI);
  System.out.printf("Right justified: |%8.4f|\n", Math.PI);
  System.out.format("Left justified:  |%-8.2f|\n", Math.PI);
  System.out.format("Right justified: |%8.2f|\n", Math.PI);
  System.out.format("Left justified:  |%-8b|\n", true);
  System.out.format("Right justified: |%8b|\n", true);
  ```

-
  ```
  Left justified:  |abc     |
  Right justified: |     abc|
  Left justified:  |25      |
  Right justified: |      25|
  Left justified:  |3.1416  |
  Right justified: |  3.1416|
  Left justified:  |3.14    |
  Right justified: |    3.14|
  Left justified:  |true    |
  Right justified: |    true|
  ```

But wait…there's more

- We have just scratched the surface of the Scanner and Formatter classes
- See the Java API for more details

The End