CS61B Lecture #13

Administrative:

- Before Project #1 due date, will run auto-grader tests *Monday night only* (sometime after midnight). If you get something submitted by then, you'll see the results of our testing on it. You can still resubmit until deadline.
- Otherwise (if you finish at the last minute), you aren't penalized, but you'll have to rely on your own testing.

Today:

Readings for Today: *Blue Reader*, §5

Readings for next Topic: *Yellow Reader*, Chapter 1.

String as a Simple Sequence Type

- Strings are essentially arrays of characters (with lots of specialized methods) that are *immutable:* cannot be changed.
- The assignments

```
String S1 = "Hello, world";
String S2 = S1;
S1 = "Goodbye, world";
```

  change what string `S1` is pointing to, but nothing you do to `S1` can change the String object that `S2` points to.
- All instance variables in a `String` are private and no method changes the visible state of a `String` object.
- So programming with Strings is inherently functional (in the sense of CS61A).

A Question of Efficiency

- In general, the time required to concatenate two strings, as in

```
String msg = prefix + suffix;
```

  requires time proportional to `msg.length ()`: must create a new String and then copy the contents of `prefix` and `suffix` into it.

- There are special cases: when can this concatenation be done much faster?

- Roughly how long does the following take?

```
String r;
r = "";
for (int i = 0; i < N; i += 1)
    r += " ";   // Same as r = r + " " (just as for numbers).
```

- Better to use `StringBuilder` to build it: a kind of modifiable String:

```
StringBuilder b = new StringBuilder ();
for (int i = 0; i < N; i += 1)
    b.append (' ');  // Takes constant amount of time "sort of"
String r = b.toString ();
```

  This all requires time proportional to `N`.

A Question of Efficiency

- In general, the time required to concatenate two strings, as in

```
String msg = prefix + suffix;
```

  requires time proportional to `msg.length ()`: must create a new String and then copy the contents of `prefix` and `suffix` into it.

- There are special cases: when can this concatenation be done much faster? A: When one string is empty.

- Roughly how long does the following take?

```
String r;
r = "";
for (int i = 0; i < N; i += 1)
    r += " ";   // Same as r = r + " " (just as for numbers).
```

- Better to use `StringBuilder` to build it: a kind of modifiable String:

```
StringBuilder b = new StringBuilder ();
for (int i = 0; i < N; i += 1)
    b.append (' ');  // Takes constant amount of time "sort of"
String r = b.toString ();
```

  This all requires time proportional to `N`.

A Question of Efficiency

- In general, the time required to concatenate two strings, as in

```
String msg = prefix + suffix;
```

  requires time proportional to `msg.length ()`: must create a new String and then copy the contents of `prefix` and `suffix` into it.

- There are special cases: when can this concatenation be done much faster? A: When one string is empty.

- Roughly how long does the following take? A: $\propto N^2$.

```
String r;
r = "";
for (int i = 0; i < N; i += 1)
    r += " ";   // Same as r = r + " " (just as for numbers).
```

- Better to use `StringBuilder` to build it: a kind of modifiable String:

```
StringBuilder b = new StringBuilder ();
for (int i = 0; i < N; i += 1)
    b.append (' ');  // Takes constant amount of time "sort of"
String r = b.toString ();
```

  This all requires time proportional to `N`.

Idea: Pattern-Driven Output

- Java's `Formatter` class used by `PrintStream.printf` and `String.format` to produce a String from a pattern String and arguments.
- A `Formatter` is essentially like a `PrintStream` (like `System.out`). To create one, say where the things you format with it are to go:

```
Formatter buildString = new Formatter ();  // Builds a String
Formatter buildFile = new Formatter (nameOfFile); // Sends to file.
```

  then use as we have been using `System.out`:

```
buildString.format ("The value of %d + %d is %d%n", x, y, x+y);
buildFile.format ("%s is %6.2f ft. tall%n", p.name, p.height);
```

  and finally extract or finalize the result:

```
buildFile.close ();
return buildString.toString ();
```

Format Specifiers

- General form ([] indicates optional):

  %[*argument num*$][*flags*][*width*][.*precision*]*conversion*

- Tell which of the arguments to convert (usually defaulted), what optional format to use, minimum number of characters, maximum number of characters or number of decimal places, and what kind of conversion is desired.
- Examples:

```
String.format ("%d", 1000) =>  "1000"     String.format ("%(d", 10) => "10"
String.format ("%5d", 1000) => " 1000"    String.format ("%(d", -10) => "(10)"
String.format ("%05d", 1000) => "01000"   String.format ("%,d", 1000) => "1,000"
```

- The %s conversion is a kind of "general" conversion. The `.toString()` method is called (if non-null). Allows you to format almost anything.

Interesting Design Points

- `Formatter` and associated classes demonstrate several interesting uses of OOP.
- Use of `.toString`.
- The target (where the characters go) is more general than `String` or file. Can call `new Formatter(T)` as long as `T` implements `Appendable` (basically: as long as it has a couple of basic `append` methods).
- The effect of `%s` can be customized. If an argument to be converted by `%s` has a type that implements `Formattable`. This interface requires a method

  ```
  void formatTo (Formatter fmt, int flags, int width, int precision);
  ```

  If the value of `x` implements this method, then `out.printf ("%#5.2s", x)` will call it to format `x`, giving it the width (5), precision (2), and flags (an encoding of #).
- I'm guessing that `Formatter` probably uses **instanceof** here:

  ```
  if (value instanceof Formattable)
     ((Formattable) value).formatTo (...);
  ```

New Topic: Patterns

- *A regular expression* describes a *set of strings*.

`x`	means	`{ "x" }`
`[ac-f]`	means	`{ "a", "c", "d", "e", "f" }`
`\d`	means	`{ "0", "1", ..., "9" }`
`yes|no`	means	`{ "yes", "no" }`
`(yes,)*`	means	`{ "", "yes,", "yes,yes,", "yes,yes,yes", ...}`
`(yes,)+`	means	`{ "yes,", "yes,yes,", ... }`

- In Java, the `Pattern` and `Matcher` classes (used by `Scanner`, `String`, and others) give you regular expressions, which allow you to describe and parse Strings.
- One example (you'll be doing stuff in lab, too): convert all instances of "<r,c>" in the String `S` to <c,r>", where r and c are integers:

```
S = S.replaceAll ("<(\\d+),(\\d+)>", "<$2,$1>");
```