Working With Text

ISE 108: Introduction to Programming

Stony Brook University

Text and Strings

- Goal: display information besides shapes and images
- We can do this now with print() and println()
  - Problem: that output only appears at the bottom of the sketchpad, where it's hard to read
  - Solution: find a way to add text to the program window

Fonts and Text

- Before we can display any text, we need to tell Processing what font to use
- A *font* is a combination of:
  - a typeface (ex. Times New Roman)
  - a style (ex. bold or italic)
  - a point size (ex. 48 point)

Fonts and Processing

- Use the `PFont` type to represent a font
- Processing needs to work with a special font format
- We can convert fonts to this format (.vlw) either before or while we run the program
  - If we create the font first, we need to store it in the sketch's "data" folder

.vlw Font Creation

- Use the Tools menu in Processing
  - Choose "Create font..." from the menu
  - Select a font and size from the list
- Creating a font beforehand allows you to work with fonts that the user may not have installed

Loading Fonts

- To use a pre-existing .vlw font in a Processing sketch, use `loadFont()`
  - `loadFont()` takes the name of the font file as its argument
  - `loadFont()` returns a `PFont` object

```
PFont font =
     loadFont("FFScala-32.vlw");
```


Creating Fonts at Run-time

- To create a font while your sketch is running, use `createFont()`
- `createFont()` requires the font name, the point size, and a boolean (indicating whether the font should be smoothed)

```
PFont f =
createFont("Courier", 24, true);
```


Font Creation Pitfalls

- Problem: you can only create a font "on the fly" if the user has it installed
- Solutions:
  1. store an original copy of the file (in .ttf or .otf) format in the "data" folder
  2. use `PFont.list()` to get a list of fonts installed on the user's computer

Using Fonts

- The `textFont()` command tells Processing what font to draw with
  - This is like `fill()` or `stroke()` for drawings
- `textFont()` takes a `PFont` as input
- Processing will use that font until you call `textFont()` again to change it

Displaying Text

- Use the `text()` command to actually display text on the screen
  - `text()` uses the font set by `textFont()`
- `text()` takes three arguments:
  - A `String` (character sequence) to print
  - the x and y coordinates where the text should be displayed


```
// Load the font to use (24-point Courier,
// with anti-aliasing turned on)
PFont f = createFont("Courier", 24, true);

// Tell Processing to use this font
// from now on
setFont(f);

// Display text starting at (25, 50)
text("Hello, world!", 25, 50);
```

Aligning Text

- Processing lets you change where text is positioned
- `textAlign()` takes a constant as its input:
  - Possible values: `LEFT`, `RIGHT`, `CENTER`
  - Call `textAlign()` *before* `text()`
- Ex. `textAlign(CENTER);`

Strings and Things

- The `text()` command takes a `String` as its first argument
  - A `String` holds a sequence of characters
- `String` is one of the most frequently used data types in Processing (and Java)
- `String` has a number of useful helper methods

String Methods

- `String()` — creates a new `String`

```
String s = new String("Hello!");
```

- Shorthand: `String s = "Hello!";`
  - This shorthand *only* works for `String`!
- `length()` — returns the total number of characters in the `String`

String Adjustments

- `trim()` — returns a new `String` with no leading or trailing whitespace
- `toLowerCase()`/`toUpperCase()` — return a *new copy* of the `String` in lower/uppercase
- All three methods leave the original `String` unchanged
  - Java strings are *immutable*

Extracting Data From Strings

- The positions in a `String` are numbered from 0 to (length - 1)
- `charAt()` — returns the character at a given position (*index*)
- `indexOf(`*str*`)` — returns the first index at which *str* occurs in the string (or -1 if it isn't there)

Extracting Data From Strings, cont'd

- `substring(start, end)` — returns a new `String` containing the characters from position *start* up to (but not including) *end*
- You can also call `substring()` with exactly one argument
  - In this case, it returns everything from the specified index through the end

String Equality

- What makes two `Strings` equal?
  - Do they have the same length? The same case? The same characters?
- According to Processing,

  "abcdef" == "abcdef"

  is `false` (they are *not* the same).
  - What gives?

Objects & Primitives

- Remember the difference between primitive (built-in) types and objects
  - Primitive variables hold an actual value
  - Object variables (*references*) only hold the address of an object!
  - This causes problems when we try to compare two objects

Processing is Shallow

- Processing performs *shallow comparisons* by default (using the == operator)
- A shallow comparison looks at the value immediately associated with a variable
  - This is okay for primitive types
- For objects, this means that we compare their *memory addresses*, not their contents!
  - Objects are only "equal" if they live at the same memory address

String Equality

- `String` has methods to test equality based on *content*, not memory location
- `equals()` — returns `true` if two `Strings` have the same sequence of characters
  - Usage: *firstString*.equals(*secondString*)
  - `equals()` requires both strings to have identical capitalization

Comparing Strings

- `compareTo()` — compares two strings for their relative lexicographical ordering
  - case, then alphabetical, then by length
- Usage: *firstString*.compareTo(*secondString*)
  - This method returns an integer value

Result	Positive	0	Negative
Meaning	first > second	first == second	first < second

Don't Be So Sensitive!

- Problem: `equals()` and `compareTo()` are case-sensitive
  - Sometimes, we only want to compare two `Strings` by length and/or characters
- Processing has two more methods for this situation:
  - `equalsIgnoreCase()`
  - `compareToIgnoreCase()`

Splitting Strings

- Use **`split()`** to break up a String into an array of shorter Strings
  - NOTE: **`split()`** is part of Processing, not the **`String`** class!
- Usage: split (*string_to_split*, *delimiter*)
  - *delimiter*: the marker that separates "words"
- Ex. split("abc def ghi jkl mno", " ") returns {"abc", "def", "ghi", "jkl", "mno"}

Reading From A File

- Use Processing's `loadStrings()` function to read the contents of a text file
- Usage: loadStrings(filename);
  - Put the file in your sketch's "data" folder
- loadStrings() returns an array of Strings
  - Each line from the file is a separate String

Saving Data to a File

- Use Processing's `saveStrings()` function
- Usage: saveStrings(*filename*, *array_of_data*)
  - *array_of_data* is an array of Strings
  - Each array element becomes a new line in the file
- WARNING: If *filename* already exists, it will be replaced/overwritten!

Getting Online Data

- `loadStrings()` is also Internet-enabled
  - Instead of a local filename, pass in a URL instead
  - `loadStrings()` will retrieve the data from the URL and save it to an array of Strings
- Ex. `loadStrings("http://www.cnn.com/index.html")`
- If you know what the format of the result is, you can then use `String` methods like `indexOf()` and `substring()` to parse it into usable pieces