CS 106 Winter 2016
Craig S. Kaplan

Module 10
Text Processing

Topics

- Useful `String` functions
- Useful `Character` functions
- Introduction to regular expressions

Readings

- The first part of Shiffman's online notes about data
- *Learning Processing*, Sections 17.1, 17.2 and Chapter 18

Introduction

We've already dealt with text in bits and pieces throughout the term. But processing text is an important enough real-world topic for lots of small programs that it's worth investigating it in more detail in its own module.

In some sense, text is the lowest common denominator of ways to represent data electronically. Sure, "binary" files are more common for media because they're much smaller and faster to process. But text files are more "accessible"—easier to read through by eye (if necessary), and easier to hack with simple scripts.

Strings and characters

We've already used a few string-related functions repeatedly during lectures, assignments and labs:

- `loadStrings()`: given a filename, retrieve the corresponding text file and break it into an array of `Strings`, one per line in the file.
- `split()`: given a string and a delimiter character or string, break the string into an array of smaller strings corresponding to sequences of characters (including empty sequences) that are separated by exact copies of the delimiter.

There are a couple of additional global Processing functions that are worth knowing about in this context:

- `splitTokens()`: This function is similar to `split()`, but arguably more intelligent. It takes two arguments as input: a `String` to process, and another `String` containing all possible delimiters that can separate "tokens". The tokens are non-empty strings that don't contain any of the delimiters, separated by any number of delimiter characters. It may take a few experiments to see why this is different than `split()`.
- `join()`: The spiritual opposite of `split()`. Given an array of strings and a separator string, return a single long string with all the strings in the original array strung together, interleaved with copies of the separator:

```
String[] cat_names = {
  "Ginkgo", "Ginseng", "Arlo",
  "Otis", "Titania" };
println( join( cat_names, ", " ) );
```

- `trim()`: return a new copy of the input string in which any leading or trailing whitespace has been removed. Useful to whittle a string down to the "important" bit.

Of course, that's not the whole story for String instances. Processing Strings are really just Java Strings, meaning that you have all the power of the underlying String class. If you're processing some text, it may be useful to browse through the Java API documentation for String in order to become acquainted with the tools that are built in. I will call attention to two in particular:

- equals(): If you want to compare two String instances, you can't just use ==, as you would with ints or floats. The worst part is that this will sometimes work, and occasionally fail without warning, meaning that if you accidentally use == it may be very difficult to spot your bug. Instead, you should always use the equals() method:

```
void sayHello( String name )
{
  // name == "Craig" will probably
  // not work.
  if( name.equals( "Craig" ) ) {
    println( "Get lost!" );
  } else {
    println( "Well hello there, "
      + name + "!" );
  }
}
```

- charAt(): A String behaves a lot like an array of characters. But it *isn't* an array, and so we've never been able to use the usual array syntax, such as square brackets to access individual characters. Instead, Strings support that behaviour through dedicated methods. In particular, charAt() takes an integer index as input and returns the character at the corresponding position in the String. The index starts at zero, just as it would for an array.

Here are a few additional String methods to have a look at; for the most part, I hope the behaviour is clear from the name.

- `startsWith()`, `endsWith()`
- `substring()`
- `toLowerCase()`, `toUpperCase()`
- `toCharArray()`

The `charAt()` method returns a value of type `char`, a type we haven't dealt with much in this course. A `char` represents a single character of text. It used to be a very simple type, kind of like a small `int`. But that was back in the days when there were only 256 possible characters to deal with; nowadays, with Unicode, a `char` needs to be capable of representing a much larger number of possible characters.

Because characters are a bit like numbers, you can compare them in fairly obvious ways. If `c1` and `c2` are both of type char, then you're allowed to ask questions like `c1 == c2`, `c1 < c2`, `c1 >= c2`, and so on. For the most part these questions will behave sensibly (e.g., if `c1` and `c2` are lowercase letters, then `c1` will be less than `c2` if `c1` comes before `c2` in the alphabet).

The main observation I want to make about `char` is that Java includes a class called Character that includes a bunch of useful static methods for determining what flavour of character you've got. Each of the following methods takes a single `char` as input and returns a `boolean`:

- `isAlphabetic()`
- `isDigit()`
- `isLetter()`
- `isLetterOrDigit()`
- `isLowerCase()`, `isUpperCase()`
- `isWhitespace()`

Keep in mind that static methods behave a bit differently than the methods we're used to in classes. They belong *to the class itself*, and not to any specific instance. You'd call them as in this example:

```
int countDigits( String inp )
{
  int total = 0;
  for( int idx = 0; idx < inp.length(); ++idx ) {
    char c = inp.charAt( idx );
    if( Character.isDigit( c ) ) {
      ++total;
    }
  }
  return total;
}
```

This is another place where the Java origins of Processing show through, which is too bad. We've seen static methods once or twice during the term. Another example is `Fisica.init( this )`, which sends the `init()` message to the `Fisica` class and not to any specific instance.

Case study: spell checker

As an example that combines many of these methods and ideas, let's write a sketch that behaves like a spell checker. First, it will read in a word list and process it so that we can check if a given `String` is a legal word. Then we'll read in some arbitrary text file and check all its words.

First, let's find a word list. We'll use SOWPODS, the standard word list for Scrabble tournament play in many countries. You can download the word list as a single enormous text file, with one word per line. Scrabble doesn't permit one-letter words, so it's important to add two words to this list: "a" and "I"!

It isn't too hard to load this list into an array of strings, and to write an `isWord()` function that tells you if a given `String` is in the list:

```
String[] wordlist;

boolean isWord( String wd )
{
  // return dict.hasKey( wd );
  for( int idx = 0; idx < wordlist.length; ++idx ) {
    if( wd.equals( wordlist[idx] ) ) {
      return true;
    }
  }
  return false;
}

void setup()
{
  wordlist = loadStrings( “sowpods.txt” );
}
```

There’s an in-between approach that I’m skipping, where you store the words in an array but search the array quickly using *binary search*. That’s definitely an advanced CS topic.

There is a downside with this approach: it can be slow. The SOWPODS list has over 250,000 words in it, and we’ll potentially search the entire array every time we check spelling. This is a case where using the “dictionary” data shape can be helpful. Recall that dictionaries are a way to map a set of keys to associated values. In this case, we don’t care about the values, just whether or not a given key (word) is in the dictionary. We can achieve this behaviour with the `IntDict` class, choosing an arbitrary value (say, 1) to associate with every legal word.

```
IntDict dict;

boolean isWord( String wd )
{
  return dict.hasKey( wd );
}

void setup()
{
  String[] wds = loadStrings( "words.txt" );
  dict = new IntDict();
  for( int idx = 0; idx < wds.length; ++idx ) {
    dict.set( wds[idx], 1 );
  }
}
```

This approach is a bit advanced, but hopefully the intended behaviour is clear. The fact that it's *far* more efficient than using an array is also an advanced topic, but if you're doing something similar with text processing it'll be a useful sample to draw from.

Note also that to the outside world, the `isWord()` function didn't change at all. Anybody who calls that function can expect to pass in the same sort of information, and get the same sort of answer in reply. It's only internally that things changed, with a vast improvement in performance. This is a recurring theme in computer science. We hide the implementation of something behind an interface, a kind of customer service desk where you make your request. You don't need to know how that request is fulfilled, which gives us the freedom to swap in different implementations as needed.

With this dictionary in place, we can read any text file line by line using `loadStrings()`, break every line into words using `splitTokens()`, and check every word against SOWPODS using `isWord()`. In reality, though, there are a few messy things that we'll want to account for as part of a spell checker:

- We'll probably want to convert the word list, and every word in the input, to lower case. That'll make words easier to look up consistently.
- What about words that have punctuation attached?
- What about possessives—should we strip off the 's?
- What about hyphenated terms?
- A word that doesn't appear in the dictionary will be reported every time it's discovered in the text. Can we consolidate our answers so that every illegal word appears once?

Even a conceptually simple tool like a spell checker can grow to be fairly complicated (more complicated than we'll address in this course) once we try to account for the annoying "corner cases" of the English language.

Example sketch: SpellCheck

As long as we're playing with building a dictionary of English words based on Scrabble, it's also interesting to write a sketch that can cheat at Scrabble: given seven letters, find all possible words of two or more letters that can be formed from subsets of them. The tricky part here is the code to generate all possible words. It turns out that the best approach is recursive! Given a partial word we've generated so far, and a set of letters we have yet to use, try all possible ways to add one of the new letters onto the partial word, and continue recursively with a longer partial word and fewer unused letters. At every stage, check the partial word to see if it's in the dictionary.

Example sketch: Scrabble

Regular expressions

https://xkcd.com/208/

Regular expressions are one of the gold standard tools that every hacker has in their utility belt. They are amazingly powerful and versatile, and handy in a wide range of text processing applications. But they're also complicated: a regular expression is basically a program in its own little programming language, and it's a terrible, ugly language to boot. There are whole books and websites devoted to the topic. We can't possibly go into that much detail in this course. So my goal is simply to introduce the topic of regular expressions and show a few examples, hoping that some day you may find it useful to have seen a first look.

The `String` method `contains()` can be used to determine if one string includes another one as a substring.

```
String s = "Call me Ishmael";
println( s.contains( "shma" ) );      // true
println( s.contains( "Pancakes!" ) ); // false
```

That's great if you know exactly what you're looking for. But what if you only know the general *form* of what you're looking for? For example, what if you wanted to search a large volume of text for a phone number? You might know that the phone number will take the form `(NNN) NNN-NNNN` where `N` in this case represents a digit from 0 to 9. But you can't use the `contains()` method here, because you don't know which exact phone number to look for. It's no use to try searching for all possible phone numbers, since there are billions of them. You're more or less stuck writing custom code from scratch. Perhaps you'd walk through the text character by character looking for an open parenthesis, and then walk forward from there checking if the subsequent text follows the pattern of a phone number.

This is where regular expressions demonstrate their true power. A regular expression is a description of a pattern that might appear in text. If the pattern does appear, we call it a *match*. The simplest question we can ask, then, is whether a given piece of text has a match for a given pattern, using the built-in `match()` function. That function returns a value of type `String[]`, which we'll get into shortly. For now, it suffices to know that if the pattern doesn't appear in the text, `match()` will return `null`.

Every regular expression is encoded in a string, and the simplest expressions are just literal strings that we want to search for:

```
String text =
  "Call me Ishmael. Some years ago--never mind" +
  " how long precisely--having little or no money" +
  " in my purse, and nothing particular to" +
  " interest me on shore, I thought I would" +
  " sail about a little and see the watery" +
  " part of the world.";

if( match( text, "I thought I" ) != null ) {
  println( "That's in the text!" );
}
```

But matching a literal piece of text is just the beginning. In most implementations of regular expressions it's possible to match classes of characters (e.g., letters, digits, whitespace), repetitions of simpler patterns, sub-patterns that may or may not appear, and on and on. For example, we could check whether a piece of text contains a Canadian postal code by using character ranges in square brackets:

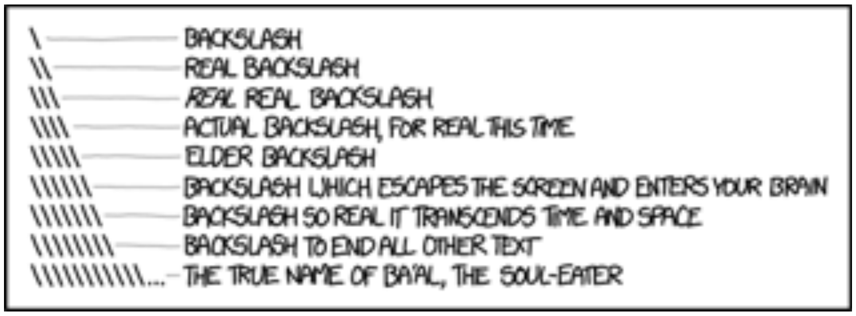
```
String pat = "[a-zA-Z][0-9][a-zA-Z] [0-9][a-zA-Z][0-9]";

boolean containsPostalCode( String str )
{
  return match( str, pat ) != null;
}

String[] tests = {
    "H0H 0H0",
    "Merry Christmas",
    "Look at this: {N2L 3G1}",
    "H9B 2H"
};

void setup()
{
  for( int idx = 0; idx < tests.length; ++idx ) {
    if( containsPostalCode( tests[idx] ) ) {
      println( tests[idx] );
    }
  }
}
```

Note the space in the pattern `pat`. That pattern won't match a postal code with no intervening space, or with a hyphen, etc., though we could add those in as alternatives We might use a similar idea to determine whether a piece of text contains a phone number, or an email address, or a date and time, or many other common patterns for text.

And what if we want to extract the actual block of text that matched the pattern? It's fine to know that text contains a phone number, but what's the actual number? That's where we can make use of the return value of `match()`. The trick is to surround relevant bits of the pattern inside parentheses. These get extracted as *groups*, and included in the array returned by `match()`.

Example sketch: FindPhoneNumbers

The full syntax of the language of regular expressions is far too complicated to present here. The version supported by Processing is precisely the language built in to Java, which is described in the Java documentation. There are also many gentle introductions to be found online. When following any of these introductions, be careful with backslashes. Remember that backslashes inside of strings are treated specially by Processing in order to communicate the backslash through Java to the underlying regular expression engine, we have to force Java not to remove it from the string. The solution is to use a *double* backslash whenever the language of regular expressions calls for a single one. What a drag!


\ BACKSLASH
\\ REAL BACKSLASH
\\\ REAL REAL BACKSLASH
\\\\ ACTUAL BACKSLASH, FOR REAL THIS TIME
\\\\\ ELDER BACKSLASH
\\\\\\ BACKSLASH WHICH ESCAPES THE SCREEN AND ENTERS YOUR BRAIN
\\\\\\\ BACKSLASH SO REAL IT TRANSCENDS TIME AND SPACE
\\\\\\\\ BACKSLASH TO END ALL OTHER TEXT
\\\\\\\\\\\\... THE TRUE NAME OF BA'AL, THE SOUL-EATER

https://xkcd.com/1638/