String Processing

Eric Roberts
CS 106A
February 3, 2010

Enigma

Cryptography at Bletchley Park

I have twice taught courses at
Stanford in Oxford when we have
visited Bletchley Park, which served
as the headquarters for the British
decryption effort during the war.
The museum at Bletchley contains
working models of the decryption
machines designed by Alan Turing,
just as they appeared in the Enigma
trailer.
The first time around, we were lucky
to have Jean Valentine, who worked
in Hut 8 during the war, as our host
at Bletchley.

Stanford’s Contribution to Cryptography

• Stanford has always been in the forefront
  of cryptographic research. In 1976,
  Professor of Electrical Engineering Martin
  Hellman and his students Ralph Merkle
  and Whitfield Diffie developed public-key
  cryptography, which revolutionized the
  process of coding messages.
  • Although Hellman, Diffie, and Merkle
    were granted a U.S. patent for their work,
    it turns out that much of the same technology
    was invented in England by the successor
    to the Government Code and Cipher
    School at Bletchley Park. That work,
    however, remained classified until the
    1990s and had no commercial impact.

Encryption

public void run() {
    public void run() {

private String encodeCaesarCipher(String str, int key) {
    if (key < 0) key = 26 - (-key % 26);
    String result = "";
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        if (Character.isUpperCase(ch)) {
            ch = (char) ('A' + (ch - 'A' + key) % 26);
        }
        result += ch;
    }
    return result;
}

Creating a Caesar Cipher

This program implements a Caesar cipher.
Character positions to shift: 11
Original message: MMEREDIFXPPN
Encoded message: NAMNCHOKCVF

Exercise: Letter Substitution Cipher

One of the simplest types of codes is a letter-substitution cipher, in which each letter in the original message is replaced by some different letter in the coded version of that message. In this type of cipher, the key is often presented as a sequence of 26 letters that shows how each of the letters in the standard alphabet are mapped into their enciphered counterparts:

```
ABCDEF GHJKLM NOPQRSTUVWXYZ
```

Enter 26-letter key: LZDRXPEAJ YBQWFVIHCTGNOMKSU

Plaintext: LEWIS CARROLL

Ciphertext: QXMJT DLCCVQQ

A Case Study in String Processing

Section 8.5 works through the design and implementation of a program to convert a sentence from English to Pig Latin. At least for this dialect, the Pig Latin version of a word is formed by applying the following rules:

1. If the word begins with a consonant, you form the Pig Latin version by moving the initial consonant string to the end of the word and then adding the suffix *ay*, as follows:

   - scram — amcra

2. If the word begins with a vowel, you form the Pig Latin version simply by adding the suffix *way*, like this:

   - apple — appleway

Starting at the Top

- In accordance with the principle of top-down design, it makes sense to start with the `run` method, which has the following pseudocode form:

  ```java
  public void run() {
    Tell the user what the program does.
    Ask the user for a line of text.
    Translate the line into Pig Latin and print it on the console.
  }
  ```

- This pseudocode is easy to translate to Java, as long as you are willing to include calls to methods you have not yet written:

  ```java
  public void run() {
    println("This program translates a line into Pig Latin.");
    String line = readLine("Enter a line: ");
    println(translateLine(line));
  }
  ```

Designing `translateLine`

- The `translateLine` method must divide the input line into words, translate each word, and then reassemble those words.

- Although it is not hard to write code that divides a string into words, it is easier still to make use of existing facilities in the Java library to perform this task. One strategy is to use the `StringTokenizer` class in the `java.util` package, which divides a string into independent units called *tokens*. The client then reads these tokens one at a time. The set of tokens delivered by the tokenizer is called the *token stream*.

- The precise definition of what constitutes a token depends on the application. For the Pig Latin problem, tokens are either words or the characters that separate words, which are called *delimiters*. The application cannot work with the words alone, because the delimiter characters are necessary to ensure that the words don’t run together in the output.

The `StringTokenizer` Class

- The constructor for the `StringTokenizer` class takes three arguments, where the last two are optional:
  - A string indicating the source of the tokens.
  - A string which specifies the delimiter characters to use. By default, the delimiter characters are set to the whitespace characters.
  - A flag indicating whether the tokenizer should return delimiters as part of the token stream. By default, a `StringTokenizer` ignores the delimiters.

- Once you have created a `StringTokenizer`, you use it by setting up a loop with the following general form:

  ```java
  while (tokenizer.hasMoreTokens()) {
    String token = tokenizer.nextToken();
    code to process the token
  }
  ```

The `translateLine` Method

- The existence of the `StringTokenizer` class makes it easy to code the `translateLine` method, which looks like this:

  ```java
  private String translateLine(String line) {
    String result = ""
    StringTokenizer tokenizer = new StringTokenizer(line, DELIMITERS, true);
    while (tokenizer.hasMoreTokens()) {
      String token = tokenizer.nextToken();
      if (isWord(token)) {
        token = translateWord(token);
      }
      result += token;
    }
    return result;
  }
  ```

- The `DELIMITERS` constant is a string containing all the legal punctuation marks to ensure that they aren’t combined with the words.
The translateWord Method

- The translateWord method consists of the rules for forming Pig Latin words, translated into Java:

```java
private String translateWord(String word) {
    int vp = findFirstVowel(word);
    if (vp == -1) {
        return word;
    } else if (vp == 0) {
        return word + "way";
    } else {
        String head = word.substring(0, vp);
        String tail = word.substring(vp);
        return tail + head + "ay";
    }
}
```

- The remaining methods (isWord and findFirstVowel) are both straightforward. The simulation on the following slide simply assumes that these methods work as intended.

The PigLatin Program

```java
public void run() {
    println("This program translates a line into Pig Latin.");
    String line = readLine("Enter a line: ");
    println( translateLine(line) );
}
```