Eric Roberts
CS 106A

Handout #30
April 30, 2012

String Processing

String Processing

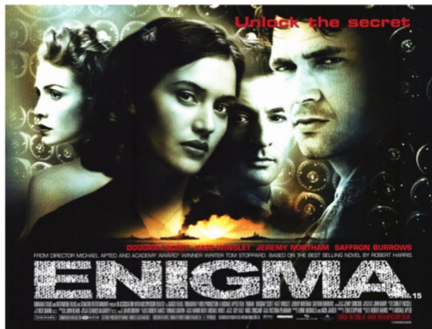
Eric Roberts
CS 106A
April 30, 2012

Alan Turing

- This June marks the 100th anniversary of the birth of Alan Mathison Turing, who made many important contributions to computer science, in areas ranging from the theory of computation, hardware design, and artificial intelligence.
- During World War II, Turing headed the mathematics division at Bletchley Park in England, which broke the German Enigma code.
- Tragically, Turing committed suicide in 1954 after being convicted on a charge of "gross indecency" for homosexual behavior. Prime Minister Gordon Brown issued a public apology in 2009.

Alan Turing (1912-1954)

Enigma



Encryption

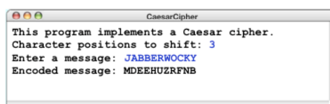
Twas brillig, and the slithy toves,
Did gyre and gimble in the wabe:
All mimsy were the borogoves,

Twas brillig, and the slithy toves,
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

Creating a Caesar Cipher

```
public void run() {
   println("This program implements a Caesar cipher.");
   int key = readInt("Character positions to shift: ");
   String plaintext = readLine("Enter a message: ");
   String ciphertext = encodeCaesarCipher(plaintext, key);
   println("Encoded message: " + ciphertext);
}
```

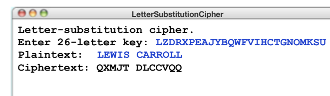
key	plaintext	ciphertext
3	JABBERWOCKY	MDEEHUZRFNB



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:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
| | | | | | | | | | | | | | | | | | | | | | | | | |
L Z D R X P E A J Y B Q W F V I H C T G N O M K S U
```



Starting at the Top

- The principle of top-down design suggests starting with the **run** method, which has the following pseudocode form:

```
public void run() {
   Tell the user what the program does.
   Ask the user to enter a 26-letter key.
   Ask the user for a line of text.
   Call a method to encrypt the line using the specified key.
   Print the result of that encryption.
}
```

- This pseudocode is easy to translate to Java:

```
public void run() {
   println("Letter-substitution cipher.");
   String key = readKey();
   String plaintext = readLine("Plaintext:  ");
   String ciphertext = encrypt(plaintext, key);
   println("Ciphertext: " + ciphertext);
}
```

- All that remains is to write the code for **readKey** and **encrypt**.

Translating Pig Latin to English

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

1. If the word begins with a consonant, the **translateWord** method moves the initial consonant string to the end of the word and then adds the suffix *ay,* as follows:

   scram ⟶ *amscray*

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

   apple ⟶ *appleway*

3. If the word contains no vowels at all, **translateWord** returns the original word unchanged.

Pseudocode for the Pig Latin Program

```
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.
}

private String translateLine(String line) {
   Initialize a string variable called result to hold the growing string.
   Divide the line into individual words and separating characters called tokens.
   while (there are still tokens to be processed) {
      Get the next token.
      If the token is a word, call translateWord to translate it to Pig Latin.
      Concatenate the token to the end of the result variable.
   }
}

private String translateWord(String word) {
   Find the first vowel in the word.
   If there are no vowels, return the original word unchanged.
   If the vowel appears in the first position, return the word concatenated with "way".
   Divide the string into two parts (head and tail) before the vowel.
   Return the result of concatenating the tail, the head, and the string "ay".
}
```

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:

```
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:

```
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:

```
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

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

```
private String translateLine(String line) {
```

```
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";
   }
}
```

vp	head	tail	word