Programming – Lecture 9

Strings and Characters (Chapter 8)

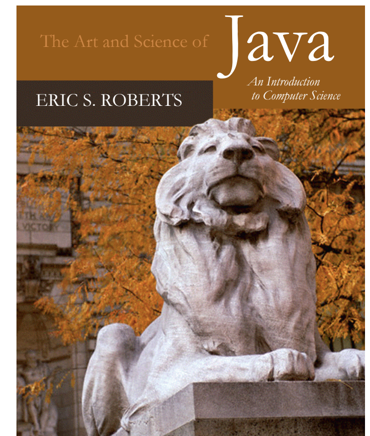
- Enumeration
- ASCII / Unicode, special characters
- **`Character`** class
- Character arithmetic
- Strings vs. characters
- **`String`** methods
- Splitting a string into tokens
- Aside: regular expressions
- Top-Down design

Five-Minute Review

1. Which three areas do we distinguish in memory? What is stored where?
2. What is *garbage collection*?
3. What is *recursion*?
4. What is a *linked list*?

CHAPTER 8

Strings and Characters



Surely you don't think that numbers are as important as words.

—King Azaz to the Mathemagician
—Norton Juster, *The Phantom Toolbooth,* 1961

8.1 The principle of enumeration

8.2 Characters

8.3 Strings as an abstract idea

8.4 Using methods in the **`String`** class

8.5 A case study in string processing

The Principle of Enumeration

- Computers tend to be good at working with numeric data. When you declare a variable of type **`int`**, for example, the Java virtual machine reserves a location in memory designed to hold an integer in the defined range.
- The ability to represent an integer value, however, also makes it easy to work with other data types as long as it is possible to represent those types using integers. For types consisting of a finite set of values, the easiest approach is simply to number the elements of the collection.
- For example, if you want to work with data representing months of the year, you can simply assign integer codes to the names of each month, much as we do ourselves. Thus, January is month 1, February is month 2, and so on.
- Types that are identified by counting off the elements are called **enumerated types**.

Enumerated Types in Java

- Java offers two strategies for representing enumerated types:
  - Defining named constants to represent the values in the enumeration
  - Using the **`enum`** facility introduced in Java 5.0
- Prior to the release of Java 5.0 in 2005, Java did not include any direct support for enumerated types. Up until then, Java programmers achieved the effect of enumerated types by defining integer constants to represent the elements of the type and then using variables of type **`int`** to store the values.
- For example, Java programmers might define names for the four major compass points as follows:

```
public static final int NORTH = 0;
public static final int EAST = 1;
public static final int SOUTH = 2;
public static final int WEST = 3;
```

- Once that definition was in place, those programmers could then store a direction constant in any integer variable.

The **enum** Facility

- Recent versions of Java support a seemingly simpler and more expressive strategy that makes it possible to define an enumeration as a distinct type. In its simplest form, the pattern for an enumerated type definition is

```
public enum name {
    list of element names
}
```

- If you use the **enum** facility, you can define a new type for the four compass points like this:

```
public enum Direction {
   NORTH, EAST, SOUTH, WEST
}
```

- You can then declare a variable of type **Direction** and use it in conjunction with the constants defined by the class.

Tradeoffs in the `enum` Facility

- The **`enum`** facility has several advantages over using integers:
  - The compiler chooses the integer codes automatically.
  - Variable declarations use a more meaningful type name.
  - The compiler checks to make sure values match the type.
  - The values of the type appear as their names when printed.
- The **`enum`** facility, however, has disadvantages as well:
  - Constants must ordinarily include the type, as in **`Direction.EAST`**.
  - Paradoxically, constants used in **`case`** clauses must *omit* this type.
  - The **`enum`** facility is actually much more complex than described here.

- Note: the on-line draft of the book does not discuss enum's in detail, as enum's where just maturing at the time of writing the draft.

Enumerated Types in Java

Strategy 1: Named constants

```
public static final int NORTH = 0;
public static final int EAST = 1;
public static final int SOUTH = 2;
public static final int WEST = 3;
int dir = NORTH;
if (dir == EAST) ...
switch (dir) { case SOUTH: ...
```

Strategy 2: ***enum*** *Types*

```
public enum Direction {
  NORTH, EAST, SOUTH, WEST }
Direction dir = Direction.NORTH;
if (dir == Direction.EAST) ...
switch (dir) { case SOUTH: ...
```

Enum Types

Are special kind of class, can also contain methods etc.:

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
...
```

Enum Types

`values()` returns array of enum values, in declared order

```
for (Planet p : Planet.values()) {
    println("Planet " + p +
            "has mass" + p.getMass());
}
```

Characters

- Computers use the principle of enumeration to represent character data inside the memory of the machine. There are, after all, a finite number of characters on the keyboard. If you assign an integer to each character, you can use that integer as a code for the character it represents.
- Character codes, however, are not particularly useful unless they are standardized. If different computer manufacturers use different coding sequence (as was indeed the case in the early years), it is harder to share such data across machines.
- The first widely adopted character encoding was ASCII (*American Standard Code for Information Interchange*).
- With only 256 possible characters, the ASCII system proved inadequate to represent the many alphabets in use throughout the world. It has therefore been superseded by Unicode, which allows for a much larger number of characters.

The ASCII Subset of Unicode

The following table shows the first 128 characters in the Unicode character set, which are the same as in the older ASCII scheme:

	0	1	2	3	4	5	6	7
00*x*	\000	\001	\002	\003	\004	\005	\006	\007
01*x*	\b	\t	\n	\011	\f	\r	\016	\017
02*x*	\020	\021	\022	\023	\024	\025	\026	\027
03*x*	\030	\031	\032	\033	\034	\035	\036	\037
04*x*	*space*	!	"	#	$	%	&	'
05*x*	(	)	*	+	,	-	.	/
06*x*	0	1	2	3	4	5	6	7
07*x*	8	9	:	;	<	=	>	?
10*x*	@	A	B	C	D	E	F	G
11*x*	H	I	J	K	L	M	N	O
12*x*	P	Q	R	S	T	U	V	W
13*x*	X	Y	Z	[	\	]	^	_
14*x*	`	a	b	c	d	e	f	g
15*x*	h	i	j	k	l	m	n	o
16*x*	p	q	r	s	t	u	v	w
17*x*	x	y	z	{	|	}	~	\177

ASCII Subset of Unicode

	0	1	2	3	4	5	6	7
00*x*	\000	\001	\002	\003	\004	\005	\006	\007
01*x*	\b	\t	\n	\013	\f	\r	\016	\017
02*x*	\020	\021	\022	\023	\024	\025	\026	\027
03*x*	\030	\031	\032	\033	\034	\035	\036	\037
04*x*	*space*	!	"	#	$	%	&	'
05*x*	(	)	*	+	,	-	.	/
06*x*	0	1	2	3	4	5	6	7
07*x*	8	9	:	;	<	=	>	?
10*x*	@	A	B	C	D	E	F	G
11*x*	H	I	J	K	L	M	N	O
12*x*	P	Q	R	S	T	U	V	W
13*x*	X	Y	Z	[	\	]	^	_
14*x*	`	a	b	c	d	e	f	g
15*x*	h	i	j	k	l	m	n	o
16*x*	p	q	r	s	t	u	v	w
17*x*	x	y	z	{	|	}	~	\177

Notes on Character Representation

- The first thing to remember about the Unicode table from the previous slide is that you don't actually have to learn the numeric codes for the characters. The important observation is that a character *has* a numeric representation, and not what that representation happens to be.
- To specify a character in a Java program, you need to use a character constant, which consists of the desired character enclosed in single quotation marks. Thus, the constant **`'A'`** in a program indicates the Unicode representation for an uppercase `A`. That it has the value 101_8 is an irrelevant detail.
- Two properties of the Unicode table are worth special notice:
  - The character codes for the digits are consecutive.
  - The letters in the alphabet are divided into two ranges, one for the uppercase letters and one for the lowercase letters. Within each range, the Unicode values are consecutive.

Special Characters

Characters that are not *printing characters*

Escape sequence: backslash + character/digits

`\b`	Backspace
`\f`	Form feed (starts a new page)
`\n`	Newline (moves to the next line)
`\r`	Return (moves to beginning of line without advancing)
`\t`	Tab (moves horizontally to the next tab stop)
`\\`	The backslash character itself
`\'`	The character `'` (required only in character constants)
`\"`	The character `"` (required only in string constants)
`\`*ddd*	The character whose Unicode value is octal number *ddd*

Methods in `Character` Class

`static boolean isDigit(char ch)` Determines if the specified character is a digit.	
`static boolean isLetter(char ch)` Determines if the specified character is a letter.	
`static boolean isLetterOrDigit(char ch)` Determines if the specified character is a letter or a digit.	
`static boolean isLowerCase(char ch)` Determines if the specified character is a lowercase letter.	
`static boolean isUpperCase(char ch)` Determines if the specified character is an uppercase letter.	
`static boolean isWhitespace(char ch)` Determines if the specified character is **whitespace** (spaces and tabs).	
`static char toLowerCase(char ch)` Converts **`ch`** to its lowercase equivalent, if any. If not, **`ch`** is returned unchanged.	
`static char toUpperCase(char ch)` Converts **`ch`** to its uppercase equivalent, if any. If not, **`ch`** is returned unchanged.	

Character Arithmetic

```
char letterA = 'a';
char letterB = letterA++;

letterB == 'B'         False
letterB == 'b'         False
letterB == 'a'         True
letterA == 'b'         True
letterA == 'B'         False

public char randomLetter() {
   return (char) rgen.nextInt('A', 'Z');
}

public boolean isDigit(char ch) {
   return (ch >= '0' && ch <= '9');
}
```

Exercise: Character Arithmetic

```
public char toHexDigit(int n) {
    if (n >= 0 && n <= 9) {
        return (char) ('0' + n);
    } else if (n >= 10 && n <= 15) {
        return (char) ('A' + n - 10);
    } else {
        return '?';
    }
}
```

Strings as an Abstract Idea

- Ever since the very first program in the text, which displayed the message **`"hello, world"`** on the screen, you have been using strings to communicate with the user.
- Up to now, you have not had any idea how Java represents strings inside the computer or how you might manipulate the characters that make up a string. At the same time, the fact that you don't know those things has not compromised your ability to use strings effectively because you have been able to think of strings holistically as if they were a primitive type.
- For most applications, the abstract view of strings you have held up to now is precisely the right one. On the inside, strings are surprisingly complicated objects whose details are better left hidden.
- Java supports a high-level view of strings by making **`String`** a class whose methods hide the underlying complexity.

Using Methods in the `String` Class

- Java defines many useful methods that operate on the **`String`** class. Before trying to use those methods individually, it is important to understand how those methods work at a more general level.
- The **`String`** class uses the receiver syntax when you call a method on a string. Instead of calling a static method (as you do, for example, with the **`Character`** class), Java's model is that you send a message to a string.
- None of the methods in Java's **`String`** class change the value of the string used as the receiver. What happens instead is that these methods *return* a new string on which the desired changes have been performed.
- Classes that prohibit clients from changing an object's state are said to be **immutable**. Immutable classes have many advantages and play an important role in programming.

Strings *vs.* Characters

- The differences in the conceptual model between strings and characters are easy to illustrate by example. Both the **`String`** and the **`Character`** class export a **`toUpperCase`** method that converts lowercase letters to their uppercase equivalents.
- In the **`Character`** class, you call **`toUpperCase`** as a static method, like this:

```
ch = Character.toUpperCase(ch);
```

- In the **`String`** class, you apply **`toUpperCase`** to an existing string, as follows:

```
str = str.toUpperCase();
```

- Note that both classes require you to assign the result back to the original variable if you want to change its value.

Strings *vs.* Characters

```
ch = Character.toUpperCase(ch);

str = str.toUpperCase();
```

Q: Why not simply:

```
str.toUpperCase();
```

A: Because Strings are immutable!

Selecting Characters from a String

- Conceptually, a string is an ordered collection of characters.
- In Java, the character positions in a string are identified by an **index** that begins at 0 and extends up to one less than the length of the string. For example, the characters in the string **`"hello, world"`** are arranged like this:

h	e	l	l	o	,		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10	11

- You can obtain the number of characters by calling **`length`**.
- You can select an individual character by calling **`charAt(`*k*`)`**, where *k* is the index of the desired character. The expression

```
str.charAt(0);
```

returns the first character in **`str`**, which is at index position 0.

Selecting Characters from a String

```
String str = "hello, world";
```

h	e	l	l	o	,		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10	11

```
int twelve = str.length();

char h = str.charAt(0);
```

Concatenation

- One of the most useful operations available for strings is **concatenation**, which consists of combining two strings end to end with no intervening characters.
- The `String` class exports a method called `concat` that performs concatenation, although that method is hardly ever used. Concatenation is built into Java in the form of the `+` operator.
- If you use `+` with numeric operands, it signifies addition. If at least one of its operands is a string, Java interprets `+` as concatenation. When it is used in this way, Java performs the following steps:
  - If one of the operands is not a string, convert it to a string by applying the `toString` method for that class.
  - Apply the `concat` method to concatenate the values.

Concatenation

Code	Output
`println("hi".concat(" there"));`	*hi there*
`println("hi" + " there");`	*hi there*
`println(0 + 1);`	*1*
`println(0 + "1");`	*01*
`println(false + true);`	*Error!*
`println("" + false + true);`	*falsetrue*
`println(false + true + "");`	*Error!*
`println(false + "" + true);`	*falsetrue*

Extracting Substrings

- The **`substring`** method makes it possible to extract a piece of a larger string by providing index numbers that determine the extent of the substring.

- The general form of the **`substring`** call is

```
str.substring(p1, p2);
```

  where **`p1`** is the first index position in the desired substring and **`p2`** is the index position immediately following the last position in the substring.

- As an example, if you wanted to select the substring **`"ell"`** from a string variable **`str`** containing **`"hello, world"`** you would make the following call:

```
str.substring(1, 4);
```

Extracting Substrings

string.**substring**(*index-first*, *index-after-last*);

```
String prog = "infprogoo".substring(3, 7);
```

Checking Strings for Equality

- Many applications will require you to test whether two strings are **equal**, in the sense that they contain the same characters.
- Although it seems natural to do so, you cannot use the **==** operator for this purpose. While it is legal to write

```
if (s1 == s2) . . .
```

the **if** test will not have the desired effect. When you use **==** on two objects, it checks whether the objects are **identical**, which means that the references point to the same address.

- What you need to do instead is call the **equals** method:

```
if (s1.equals(s2)) . . .
```

Checking for Equality

```
String s1 = new String("hello");
String s2 = new String("hello");
```

```
s1 == s2
s1.equals(s2)
```

False
True

```
String s3 = "hello";
String s4 = "hello";
String s5 = "hel" + "lo";
```

```
(s3 == s4) && (s4 == s5)
s1.intern() == s2.intern()
```

True
True

JVM uses *string literal pool*

Coding advice: use `equals()` to compare strings

Comparing Characters and Strings

- The fact that characters are primitive types with a numeric internal form allows you to compare them using the relational operators. If **`c1`** and **`c2`** are characters, the expression

```
c1 < c2
```

  is **`true`** if the Unicode value of **`c1`** is less than that of **`c2`**.

- The **`String`** class allows you to compare two strings using the internal values of the characters, although you must use the **`compareTo`** method instead of the relational operators:

```
s1.compareTo(s2)
```

  This call returns an integer that is less than 0 if **`s1`** is less than **`s2`**, greater than 0 if **`s1`** is greater than **`s2`**, and 0 if the two strings are equal.

Comparing Characters and Strings

```
char c1 = 'a', c2 = 'c';

c1 < c2
```

True

```
String s1 = "a", s2 = "c";

s1.compareTo(s2)
```

-2

Searching in a String

- Java's **`String`** class includes several methods for searching within a string for a particular character or substring.
- The method **`indexOf`** takes either a string or a character and returns the index within the receiving string at which the first instance of that value begins. If the string or character does not exist at all, **`indexOf`** returns –1. For example, if the variable **`str`** contains the string **`"hello, world"`**:

`str.indexOf('h')`	*returns*	0
`str.indexOf("o")`	*returns*	4
`str.indexOf("ell")`	*returns*	1
`str.indexOf('x')`	*returns*	–1

- The **`indexOf`** method takes an optional second argument that indicates the starting position for the search. Thus:

`str.indexOf("o", 5)`	*returns*	8

Searching in a String

```
String str = "informatik";

str.indexOf('i')              0
str.indexOf("n")              1
str.indexOf("form")           2
str.indexOf('x')             -1

str.indexOf('i',1)            8
```

Other Methods in `String` Class

`int lastIndexOf(char ch)` *or* **`lastIndexOf(String str)`** Returns the index of the last match of the argument, or −1 if none exists.
`boolean equalsIgnoreCase(String str)` Returns **`true`** if this string and **`str`** are the same, ignoring differences in case.
`boolean startsWith(String str)` Returns **`true`** if this string starts with **`str`**.
`boolean endsWith(String str)` Returns **`true`** if this string ends with **`str`**.
`String replace(char c1, char c2)` Returns a copy of this string with all instances of **`c1`** replaced by **`c2`**.
`String trim()` Returns a copy of this string with leading and trailing whitespace removed.
`String toLowerCase()` Returns a copy of this string with all uppercase characters changed to lowercase.
`String toUpperCase()` Returns a copy of this string with all lowercase characters changed to uppercase

Simple String Idioms

Iterating through characters in a string:

```
for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    . . . code to process each character in turn . . .
}
```

Growing new string character by character:

```
String result = "";
for (whatever limits are appropriate to application) {
    . . . code to determine next character to be added . . .
    result += ch;
}
```

Exercises: String Processing

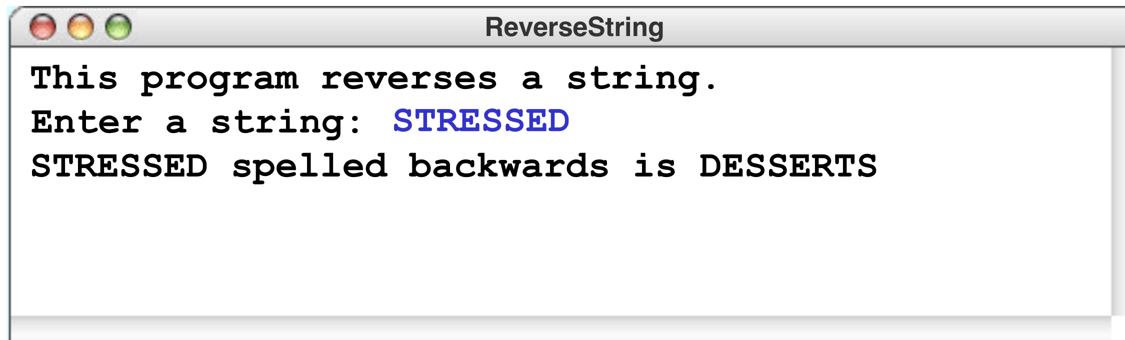
```
public String toUpperCase(String str) {
    String result = "";
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        result += Character.toUpperCase(ch);
    }
    return result;
}

public int indexOf(char ch) {
    for (int i = 0; i < length(); i++) {
        if (ch == charAt(i)) return i;
    }
    return -1;
}
```

reverseString

```
public void run() {
   println("This program reverses a string.");
   String str = readLine("Enter a string: ");
   String rev = reverseString(str);
   println(str + " spelled backwards is " + rev);
}
```

rev	str
DESSERTS	STRESSED



skip simulation

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 **`String.split()`** method, 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

- Note: an alternative to the **`String.split()`** method is the **`StringTokenizer`**. However, this is now considered **deprecated** and should not be used for new code.
- 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
}
```

Splitting a String into Tokens

Method in **`String`**: **`split()`**

Example:

```
String line = "One short line";
String[] tokens = line.split("\\s");
for (String token : tokens) {
  println(token);
}
```

produces

```
One
short
line
```

Aside: Regular Expressions

`String[] split(String regex)`

`regex` must be *regular expression* (RE)

- RE belongs to regular language (RL)
- RL is also CFL (but not necessarily the converse!)

In string literals, must use double backslashes (e.g. **`\\s`**) to encode backslash (**`\s`**)

https://www.quora.com/What-is-the-difference-between-regular-language-and-context-free-language

https://docs.oracle.com/javase/9/docs/api/java/util/regex/Pattern.html#sum

Aside: Regular Expressions

x	The character x
\\	The backslash character
\0n	The character with octal value 0n (0 <= n <= 7)
\t	The tab character ('\u0009')
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
^	The beginning of a line
$	The end of a line
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times

Pig Latin

1. If word begins with consonant, move initial consonant string to end and add suffix *ay*:

   scram → *amscray*

2. If word begins with vowel, add suffix *way*:

   apple → *appleway*

Top-Down Design

```
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.

```
}
```

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

```
private String translateLine(String line) {
  String result = "";
  String[] tokens = line.split("\\s");
    for (String token: tokens) {
      if (isWord(token)) {
      token = translateWord(token);
    }
    result += token;
  }
  return result;
}
```

```
private boolean isWord(String token) {
   for (int i = 0; i < token.length(); i++) {
      char ch = token.charAt(i);
      if (!Character.isLetter(ch))
         return false;
   }
   return true;
}
```

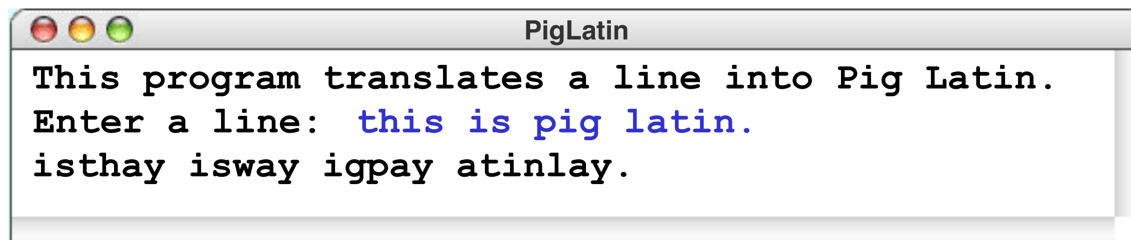
```
private String translateWord(String word) {
   int vp = findFirstVowel(word);
   if (vp == -1) {
      return word;
   } else if (vp == 0) {
      return word + "way";
   } else {
      String head = word.substring(0, vp);
      String tail = word.substring(vp);
      return tail + head + "ay";
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

PigLatin

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

line

this is pig latin.



skip simulation

`StringBuilder`

- **Recall:** Strings are immutable
- Whenever we operate on strings, e.g. with **`append()`**, must create new **`String`** objects
- More efficient, but usually less convenient, alternative to **`String`**: **`StringBuilder`**

Programming advice:

- If performance really is an issue, use **`StringBuilder`**
- Otherwise, use **`String`**

Example with **`String`**

```
String str = "";
for (int i = 0; i < n; i++) {
  str += str.length() + " ";
}
println(str);
```

produces for **`n`** = 10:

```
0 2 4 6 8 10 13 16 19 22
```

Example with `StringBuilder`

```
StringBuilder str = new StringBuilder();
for (int i = 0; i < n; i++) {
   str.append(str.length() + " ");
}
println(str);
```

produces for `n` = 10:

```
0 2 4 6 8 10 13 16 19 22
```

Aside: Measuring Execution Time

- Outcome of program is (usually) deterministic
- However, run time is not
- Issues influencing timing:
  - Memory hierarchy: instruction cache, data cache
  - Scheduling, process interference
  - Just-in-time compilation
  - I/O delays
  - ...
- Therefore, do *multiple* measurement runs

```
private static final int NUM_TRIALS = 10;
long minTime, maxTime, startTime, stopTime, elapsedTime;

minTime = maxTime = 0;
for (int trial = 0; trial < NUM_TRIALS; trial++) {
  startTime = System.nanoTime();
  String str = "";
  for (int i = 0; i < n; i++) {
    str += str.length() + " ";
  }
  println(str);
  stopTime = System.nanoTime();
  elapsedTime = stopTime - startTime;
  if (minTime == 0 || elapsedTime < minTime) {
    minTime = elapsedTime;
  }
  if (elapsedTime > maxTime) {
    maxTime = elapsedTime;
  }
}
println("One trial took " + minTime / 1e6 + " to "
   + maxTime / 1e6 + " msec.");
```

Summary I

- Principle of *enumeration*: map non-numeric properties/data (e.g. characters) to numbers
- Java provides **`enum`** types
- **`char`**/**`Character`** maps characters to *Unicode*
- Distinguish *printing characters* and *special characters*
- Express special characters with *escape sequences*

Summary II

- Conceptually, strings are ordered collections of characters
- Strings are *immutable*
- Strings should be compared with **`equals`** or **`compareTo`**, not with **`==`**
- If (!) performance is an issue, use **`StringBuilder`** instead of **`String`**
- Care should be taken when measuring execution times

https://www.youtube.com/watch?v=Hsx5R94YFAA