# Mat 2170
# Week 11

Characters and Strings

Spring 2014



## Student Responsibilities

- Reading: Textbook, Sections 8.2–8.4
- Lab: Character and String processing
- Attendance

> **Surely you don't think that numbers are as important as words.**
> King Azaz to the Mathemagician
> *The Phantom Tollbooth*, 1961, by Norton Juster

### Chapter Eight Overview

1. **Characters** - the primitive type **char**
2. Strings as an abstract idea
3. Using methods in the **String** class
4. A case study in string processing



## Characters

- The primitive type **char** can store a single character.
- There are a **finite** number of characters on the keyboard.
- **[Collating Sequence]** If we assign an **integer** to each **character**, we can use that integer as a **code** for the character it represents.
- Character codes are not particularly useful unless they are **standardized**.
- If different computer manufacturers use different coding sequences (as was the case in the "early" years), it is harder to share such data across machine platforms.



## ASCII

- The first widely adopted character encoding was **ASCII** — **American Standard Code for Information Interchange**.
- With only 256 possible characters (the number of bit combinations in a byte), 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- It has therefore been superseded by **Unicode**, which allows for a much larger number of characters.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 first 128 characters — written in Octal or **Base 8**

| *base* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 000 | \000 | \001 | \002 | \003 | \004 | \005 | \006 | \007 |
| 010 | \b | \t | \n | \013 | \f | \r | \016 | \017 |
| 020 | \020 | \021 | \022 | \023 | \024 | \025 | \026 | \027 |
| 030 | \030 | \031 | \032 | \033 | \034 | \035 | \036 | \037 |
| 040 | *space* | ! | " | # | $ | % | & | ’ |
| 050 | ( | ) | * | + | , | - | . | / |
| 060 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 070 | 8 | 9 | : | ; | < | = | > | ? |
| 100 | @ | A | B | C | D | E | F | G |
| 110 | H | I | J | K | L | M | N | O |
| 120 | P | Q | R | S | T | U | V | W |
| 130 | X | Y | Z | [ | \ | ] | ^ | _ |
| 140 | ‘ | a | b | c | d | e | f | g |
| 150 | h | i | j | k | l | m | n | o |
| 160 | p | q | r | s | t | u | v | w |
| 170 | x | y | z | { | \| | } | ~ | \177 |



## Notes on Character Representation

- There is **NO reason** to memorize underlying numeric codes for the characters
- The important observation is that each **character** has a **numeric representation** — **not** what that representation happens to be.

## Character Constants

- To specify a character in a Java program **use a character constant** which consists of the desired character enclosed in single quotation marks. **Don't use the numeric code!**
- The constant `'A'` in a program indicates the Unicode representation of an uppercase `A`.
- That an uppercase `A` has the value $101_8$ is an **irrelevant detail** — use the character, **not** the collating sequence code value.



## Properties of Unicode

**Two properties** of the Unicode table **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** of alphabetic letters, the Unicode values are **consecutive**.



## Special Characters

- Most of the characters in the Unicode table are familiar ones that appear on the keyboard.
- These characters are called **printing** (or **printable**) characters.
- The table also includes several **special characters** that are typically used to control **formatting** — for example, '\n'.
- **Special characters** are indicated by an "**escape**" sequence: a **backslash** followed by a character or sequence of digits.



## The Most Common Special Characters

| | |
|---|---|
| `\b` | Backspace |
| `\f` | Form feed — starts a new page |
| `\n` | moves to the next line — (**Newline**) |
| `\r` | Return — moves to the beginning of the current 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 the octal number `ddd` |



## The `Character` Class

- The **Character** class is defined in the **java.lang** package and is therefore available in any Java program without an `import` statement.
- The **Character** class provides several useful methods for **manipulating char** values.



## The `Character` Class

- `Character` methods are **static** — they belong to the entire class, rather than to any particular object of the class.
- These methods can be used with **char** objects — similar to **Math** class methods and how they work on numeric types.
- It is good programming practice to use these library methods rather than writing our own.

## Why Use `Character` Class Methods

- They are **standard** — programmers recognize them and know what they mean.
- Library methods have been **tested** by millions of client programmers, so it is reasonable to expect that they are **correct**.
- Implementations in the `Character` class are able to **convert characters** in alphabets other than our Roman (English) alphabet.
- Library methods are typically more **efficient** than methods we write ourselves.



## Useful Static Methods in the `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 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 or tabs.



## Two Which Return `char` Rather Than `boolean`

`static char toLowerCase(char ch)`
**Returns** a **copy** of `ch` converted to its lowercase equivalent, if any.
Otherwise, the value of `ch` is returned unchanged.

`static char toUpperCase(char ch)`
**Returns** a **copy** of `ch` converted to its uppercase equivalent, if any.
Otherwise, the value of `ch` is returned unchanged.

**Neither of these methods modifies the `char` argument sent to the method.**



## Character Arithmetic

- The fact that characters have underlying **integer representations** allows us to use them in **arithmetic expressions**.
- For example, if you evaluate the expression `'A' + 1`, Java will convert the character `'A'` into the integer 65 and then add 1 to get 66, which is the character code for `'B'`
- As an example, the following method returns a randomly chosen uppercase letter—why is the **(char)** in the return statement?

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



## Implementation of `isDigit`

- The following code implements the `isDigit()` method from the `Character` class — given a character, is it a digit between zero and nine?:

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

- Only because the numerals '0' through '9' are **consecutive** in the collating sequence could the method be written this way.



## An Exercise in Character Arithmetic

We wish to implement a method, `toHexDigit()`, that takes an integer and returns the corresponding **hexadecimal** (base 16) digit as a character:

1. If the argument is between 0 and 9, the method should return the corresponding character between '0' and '9'.
2. If the argument is between 10 and 15, the method should return the appropriate letter in the range 'A' through 'F'.
   `A = 10,  B = 11,  C = 12`
   `D = 13,  E = 14,  F = 15`
3. If the argument is outside this range, the method should return a question mark, '?'.

## One Solution

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



## Strings as an Abstract Idea

- Ever since our very first program, which displayed the message "`hello, world`" on the screen, we have been using strings to **communicate** with the user.
- Up to now, we haven't known much about how Java represents strings inside the computer, or how to manipulate them other than to use '+' for concatenation — but that hasn't stopped us from using them!
- We've been able to use strings effectively up to now because we have thought of them holistically, as if they were a **primitive** type.



- For most applications, this abstract view of strings is precisely the right one — however, on the inside, strings are surprisingly complicated objects.
- Java supports a high–level view of stings 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 we call a `String` method — unlike the `Character` class static methods which take `char`s as arguments — in Java, we **send a message** to a `String` object.



## `String` Methods **Do Not Change** `String` Objects

- **None of the methods in Java's `String` class change the value of the string used as the receiver.**
- Instead, 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 `Character` class export a `toUpperCase()` method that converts lowercase letters to their uppercase equivalents.

## toUpperCase()

- In the `Character` class, you call `toUpperCase()` as a **static method**:

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

- In the `String` class, you apply **toUpperCase()** to an existing string by **sending a message** to the string:

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

Note that both classes require us to **assign the result back to the original object** if we want to **change its value**.



## 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:

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



## The charAt() Method

- We can obtain the number of characters in a `String` object by sending it the **length()** method, which returns an `int`.
- We can select an individual character in a `String` by sending it the **charAt(k)** message, where `k` is the index of the desired character.
- The expression **str.charAt(0)** returns the first character in the `String str`, which is a index position 0.
- For example:

```
char firstChar = 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, like:

```
"life" + "jacket" --> "lifejacket"
```

- 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, which you've used multiple times in `print()` and `println()` statements, e.g.:

```
println("(" + x + ", " + y + ")");
```



## String Concatenation

- If we use "+" with numeric operands, it signifies addition.
- If **at least one operand** is a string, Java interprets "+" as concatenation.
- When it is used in this way, Java performs the following steps:
  1. If one of the operands is not a string, convert it to a string by applying the **toString()** method **for that class**.
  2. Apply the `concat()` method to concatenate the values.



## 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:

```
String newStr = str.substring(p1, p2);
```

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

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

```
String newStr = str.substring(1, 4);
```

## Checking String Equality

- We often wish to test whether two strings are equal, in the sense that they contain the same characters.
- Although it would seem natural to do so, we **cannot** use the "==" operator for this purpose.



## Checking String Equality

**Given** `s1` **and** `s2` **are** `String` **objects:**

- While it is legal to write: `if (s1 == s2) ...`, the test will **not have the desired effect**.
- When "==" is used on two **objects**, it checks whether they reference the **same memory address** — whether they are, indeed, identical down to where they are stored.
- To test for equality we must utilize the **equals()** method:

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



## Comparing Characters and Strings

- The fact that characters are primitive types with a numeric internal representation allows us 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`.



## compareTo()

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

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

- The **compareTo()** returns
  1. a **negative** value if `s1` is **less** than `s2`
  2. a **positive** value if `s1` is **greater** than `s2`
  3. **0** if the two strings are **equal**



## 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 the value begins.
- If the string or character does not exist at all in the target string, **indexOf()** returns $-1$.



## Search Examples

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

- If the `String` object `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
```

## Other `String` Class Methods

`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 diffences 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.

`toLowerCase()`
Returns a copy of this string with all uppercase characters changed to lowercase.

`toUpperCase()`
Returns a copy of this string with all lowercase characters changed to uppercase.



## Recurring String Patterns

The following two patterns are particularly important and are sometimes seen in combination:

1. **Iterating through the characters in a string**:

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

2. **Growing a new string, character by character**:

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



## Exercises: String Processing

As a **client** of the `String` class, how could we implement a new method, `toUpperCase(str)` (not part of the `String` class), so it **returns an uppercase copy of** `str`?

```
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;
}
```

Given a `String str`, how could we modify it using this method so all letters (in `str`) are uppercase?



Suppose now we are actually implementing the `String` class. How would we code the `indexOf(ch)` method (**inside** the class)?

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

Recall that since this method would be part of the `String` class, the invocations of `length()` and `charAt()` do not require a receiver, and are being sent to the `String` object itself.



## Reversing a `String` object

Suppose we need a method to reverse the characters in a string. For example, **STRESSED** reversed is **DESSERTS**

```
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);
}
```

How could we implement the `reverseString()` method?

## Pig Latin — 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**, form the Pig Latin version by moving the initial consonant substring to the end of the word, then add the suffix ay as follows:

   scram → amscray

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

   apple → appleway



## Starting at the Top — the General Algorithm

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

```
public void run()
{
    Tell the user what the program does
    Prompt for and get a line of text from the user
    Translate the line into Pig Latin and display
}
```



## Implementing the Pseudocode

- The pseudocode is easy to translate to Java, as long as we are willing to include calls to methods we haven't written yet:

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

- Now all we must do is **design** and **implement** `translateLine()`!



## Designing `translateLine()`

The `translateLine()` method must:

1. **divide** the input line into words
2. **translate** each word
3. **re–assemble** those words into a new string



## Using Built–in Classes

- But why re-invent the wheel — it is easier to make use of existing facilities in the Java library to perform the dividing task.
- One strategy is to use the **StringTokenizer** class in the `java.util` package, which divides a string into independent units, called **tokens**.
- A client of a **StringTokenizer** object then asks for these tokens one at a time.
- The set of tokens delivered by the tokenizer is called the **token stream**.



## What's a token?

- The precise definition of what constitutes a token depends on the application — what problem we're solving.
- 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 — the last two are **optional**:

1. A string which is the **source** of the tokens
2. A string which specifies the **delimiter characters** to use. By default, the delimiter characters are set to the whitespace characters.
3. A **flag** indicating whether the tokenizer should return delimiters as part of the token stream. By **default**, a StringTokenizer **ignores** the delimiters.



Creating a StringTokenizer has the general form:

```
StringTokenizer tokenizer =
    new StringTokenizer(
          inputString,        // token source
          DELIMITERS,         // token separators
          includeDelimiters); // return delimiters?
```



Once we have created a StringTokenizer, we 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

```
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 they aren't combined with the words.



## The translateWord() Method

The rules for forming Pig Latin words, rewritten in Java:

```
private String translateWord(String word))
{
    int vowelPosn = findFirstVowel(word);
    if (vowelPosn == -1) {      // no vowel, no rule
      return word;
    }
    else if (vowelPosn == 0){  // begins with vowel
      return word + "way";
    }
    else {                     // begins with consonant
      String head = word.substring(0, vowelPosn);
      String tail = word.substring(vowelPosn);
      return tail + head + "ay";
    }
}
```



## All That's Left to Do...

- Design and implement the **isWord()** method
- Design and implement the **findFirstVowel()** method

when finished, the result of entering the line

```
this is pig latin.
```

should be the String:

```
isthay isway igpay atinlay.
```