Strings
About Strings

• Strings are objects, but there is a special syntax for writing String literals: "Hello"

• Strings, unlike most other objects, have a defined operation (as opposed to a method):

  " This " + "is String " + "concatenation"

• Strings can contain any character, but some of them must be “escaped” in order to write them in a literal
  – \" stands for the double-quote (") character
  – \n stands for the newline character
  – \\ stands for the backslash (\) character
  – Each of these is written as a two-character sequence, but represents a single character in the string
Useful String methods

• **charAt(int index)**
  — Returns the character at the given index position (0-based)

• **boolean startsWith(String prefix)**
  — Tests if this String starts with the prefix String

• **boolean endsWith(String suffix)**
  — Tests if this String ends with the suffix String

• **boolean equals(Object obj)**
  — Tests if this String is the same as the **obj** (which may be any type; **false** if it’s not a String)

• **boolean equalsIgnoreCase(String other)**
  — Tests if this String is equal to the other String, where case does not matter

• **int length()**
  — Returns the length of this string; note that this is a method, not an instance variable
Some more String methods

- **int indexOf(char ch)**
  - Returns the position of the first occurrence of `ch` in this String, or `-1` if it does not occur

- **int indexOf(char ch, int fromIndex)**
  - Returns the position of the first occurrence of `ch`, starting *at* (not *after*) the position `fromIndex`

- **int lastIndexOf(char ch)**
  - Returns the position of the last occurrence of `ch` in this String, or `-1` if it does not occur

- **int lastIndexOf(char ch, int fromIndex)**
  - Returns the position of the last occurrence of `ch`, searching backward starting at position `fromIndex`
Yet some more methods

- **String substring(int beginIndex)***
  - Returns a new string that is a substring of this string, beginning with the character at the specified index and extending to the end of this string.

- **String substring(int beginIndex, int endIndex)***
  - Returns a new string that is a substring of this string, beginning at the specified `beginIndex` and extending to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex - beginIndex`.

- **String toUpperCase()***
  - Returns a new String similar to this String, in which all letters are uppercase.

- **String toLowerCase()***
  - Returns a new String similar to this String, in which all letters are lowercase.

- **String trim()***
  - Returns a new String similar to this String, but with whitespace removed from both ends.
Understanding “index”

• With `charAt(index)`, `indexOf(x)`, and `lastIndexOf(x)`, just count characters (starting from zero)
  – Similar to Python’s indexing in a String

• With `substring(from)` and `substring(from, to)`, it works better to count positions *between* characters
  – Remember: `from` is inclusive, `to` is exclusive

• Eclipse example: `StringIndices.java`
The **split** method

- **String[] split(String regex)**
  - Breaks the string up into an array of strings
  - The parameter is a *regular expression* that defines what separates the strings
- For example,
  - `String s = "one, two, three";`
  - `String[] ss = s.split("", ");`
  - This assigns the array `{"one", "two", "three"}` to `ss`
- Regular expressions allow you to capture many possible strings with a single expression
  - Regular expressions are powerful but are beyond the scope of this class
- Bottom line: use split the same way you did in Python
Strings are immutable

• A String, once created, cannot be changed
• None of the preceding methods modify the String, although several create a new String
• Statements like this create new Strings:
  \[ \text{myString} = \text{myString} + \text{anotherCharacter}; \]
• Creating a few extra Strings in a program is no big deal
• Creating a lot of Strings can be very costly (slow)
Comparing Strings

• If you write
  
  ```java
  String s = "abc";
  String t = "abc";
  ```

  the compiler only creates the string "abc" once, and makes s and t both refer to this one string
  
  – Hence, the test s == t will be true (it compares pointers)

• However, if you now write
  
  ```java
  String u = "a" + "bc";
  ```

  the test s == u will be false
  
  – This is because they are different strings (different pointers)

• Moral: Use `s.equals(u)` for strings, not ==
Characters

• Remember that `char` is a primitive type in Java
  – It is actually represented as an 8-bit number on the machine
  – You can use `chars` as numbers (add, subtract, compare with `==`, etc.)
  – Viewing `chars` as numbers might be useful sometimes

• `chars` are written with single quotes in Java
  – If you use double quotes, you get a String (even for a single character)

• Being a primitive type, `char` has no methods
  – There exists a `Character` class with some static methods
  – Will cover it if we need it

• Eclipse example: `StringAndCharExamples.java`