Strings (JFS, chap 15)

- In this section:
  - Motivation and basic `String` concepts
  - Using strings
  - String algorithms
  - String library operations

Motivation

- "Characters" correspond (roughly) to individual key strokes
  - Type name: `char` (a "primitive" type)
  - Literal values in Java: `'a'`, `'1'`, `'{'`, etc
    plus ***"escape sequences"*** such as `'\''`, `'\n'`, `'\t'`, …
- The escape sequences are needed for special characters that may be needed:
  - `\'` means a literal single quote, not end of character
  - `\t` means a tab character
  - `\r` means a carriage-return character
  - `\n` means a newline character (enter)
  - `\\` means a literal backslash, not an escape sequence (e.g. as used in a Windows file name)
- Can compare characters with `==`, `!=`, `<`, `<=`, etc

- Strings can be thought of as *arrays of characters*
- However, strings are so useful that they have many more pre-defined operations than arrays
- Strings are commonly used for:
  - Messages sent to the user, like "Please enter a number"
  - Messages entered by the user, like the name of a person
  - Textual information generally
- Strings are written in double quotes: `"abc def"`
- Escape sequences are also used in strings, especially:
  - `\"` means a literal double quote, not end of string
  - Example: `"The boy said \"Hello\"\n"`

Using Strings

- Strings can be joined (appended/concatenated) by using the `+` operator:

```
"There are " + count + " students"
```

  - This creates a *new string* from the joined parts
  - Note: the original component strings are *unchanged*
- The strings surrounding `count` have spaces so that the number is separated from the text.
- The above uses '`+`' to mean adding strings, not numbers; the operator is said to be **overloaded** (it has several meanings)
- A useful property of string concatenation is that non-string values are automatically converted to strings
- `count` in the above is an integer, but its string representation is used (e.g. "10")

- As we have seen, there are methods to convert the usual types to strings, e.g.:

```
String s = Integer.toString(42);
                // creates the string "42"
String s = Double.toString(3.14);
                // creates the string "3.14"
```

- Notice that the `toString` methods come from classes whose names begin with a capital (`Integer`, `Double`)
- (Technical detail: this is because the conversion is done by *class methods* provided by *wrapper classes*, not by the built-in types (`int`, `double`))

- Conversely, strings can be converted to the corresponding types, e.g.:

```
int i = Integer.parseInt("42");
            // creates the int value 42
double d = Double.parseDouble("3.14");
            // creates the double value 3.14
```

- The snag about converting strings to other kinds of values is that the format of the string might be wrong, e.g. "42X" is not a valid integer.
- In such a case, an **exception** (`NumberFormatException`) could stop the program.
- The programmer should *catch* this exception if the input data (e.g. numbers typed by the user) could be incorrect

Declaring and Initialising Strings

- Literal strings can be used directly in the program, e.g.:

```
g.drawString("Have A Nice Day", 20, 10);
```

- String variables can also be declared and initialised as might be expected:

```
String name1, name2;
String today = "Monday";
```

- Notice that the type ***String*** starts with a capital letter since it is the name of a library class (unlike **int**, for example, which is a built-in type)
  - So, other class/objects aspects apply too:
  - String variables hold a *pointer* to the actual string
  - There are methods used like this: `name1.length()`

Using Strings – more methods

- Strings are rather like arrays of characters
- Character positions in strings start at 0 (**not** 1)
- Like an array, strings have a *length* but calculated by a **method**:

```
"abc".length()    // Note () after length
```

yields 3

```
name1.length()
```

- The *length* is **one more than** the highest character position
- The character at a given position in a string can be extracted:

```
String s = "Java for Students";
char c = s.charAt(2);  // get character
                       at position 2
```

which will set `c` to `'v'` (counting 'J' as position 0)

Using Strings – searching

- It is possible to search a string for a given substring:

```
String s = "Java for Students";
int pos = s.indexOf("for");
          // Find where string "for" occurs
```

  which will set `pos` to `5` ('J' is position 0)

- If the substring is not found, `indexOf` returns `-1`
  - Which is never a valid character position
- It is possible to tell `indexOf` which character position to begin searching at:

```
String s = "Java for Students";
int pos = s.indexOf("t", 12);
        // Find where "t" occurs at 12 or beyond
```

  which will set `pos` to `15`

- This is useful in a loop to find *all occurrences of a substring*:

```
String s = "Java for Students";
String target = "a";
JTextArea locations = new JTextArea("");
                          // For the results
int pos = 0;              // Start at position 0
while (pos >= 0) {        // Still looking?
    pos = s.indexOf(target, pos);
                  // Look for target from position
    if (pos >= 0) {       // Target string found?
        locations.append( "" + pos + "\n");
                          // Append position found
        pos++;            // Move to next position
    }
}
```

Note

Using Strings - substring

- The *tail* of a string from a given character position can be extracted with:

```
String s = "Java for Students";
String part = s.substring(5);
                    // Get from position 5 onwards
```

  - which will set `part` to `"for Students"`
- Note: `IndexOutOfBoundsException` will be thrown if the start index is negative or larger than the length of the string

- An extra parameter can be given for the character position where extraction stops:

```
String s = "Java for Students";
String part = s.substring(5, 8);
          // From position 5, stopping at 8
```

- which will set `part` to `"for"`
- Note: This does not mean characters 5 to 8 inclusive; the end character position is *not extracted*
  - Not so strange as it seems: the end position is often the result of searching for a *delimiter*

Manipulating Strings – more methods

- It would be tempting to check if two strings are equal by writing:

```
if (name1 == name2)     // incorrect check!
  ...
```

- This actually compares the two *pointers* to `String` objects
  - So will give `false` for two separate strings even if they have the *same characters*
- We need to check strings character-by-character
  - Instead the `equals` method has to be used:

```
if (name1.equals(name2)) // names the same?
  ...
```


- Note: in `name1.equals(name2)`
  - Technically: `name1` is a string that has a method `equals` - it is called, sending parameter `name2`
  - Think of this as meaning: 'ask `name1` if it equals `name2`'
- Of course this could just as well have been written:

```
name2.equals(name1)
```

- Other typical usage:

```
name.equals("James")
name.equals("")
"James".equals(name)
```

- The `equalsIgnoreCase` method is similar, but considers upper-case and lower-case letters to be the same:

```
"Sue".equalsIgnoreCase("sUE") is true
```

- The `compareTo` method can be used to compare the 'dictionary' ('lexicographic') order of two strings:

```
int result = name1.compareTo(name2);
```

  - `result` is `< 0`: `name1` comes earlier
  - `result` is `== 0`: both strings are the same
  - `result` is `> 0`: `name1` comes later
- Examples:

```
"apple" < "cook" < "cooked" < "cooks" < "dog"
"12" < "123" < "124" < "2" < "260" < "32"
```

- If strings have mixtures of lower/upper case, `compareTo` might give unexpected results:

```
"Cook" < "aPPLE" < "aPpLE"
```

- Perhaps even stranger with digits/punctuation:

```
"1c.ok" < "1cook" < "Cook" < "[ook" < "cook"
```

Why? Next slide…

- `compareTo` actually uses the *character codes* in the strings
- On most computers the "collating sequence" of characters is:
  - punctuation (space, `!`, …, `/`) then
  - digits (`0`, …, `9`) then
  - punctuation (`:`, …, `@`) then
  - upper-case letters (`A`, …, `Z`) then
  - punctuation (`[`, …, `` ` ``) then
  - lower-case letters (`a`, . . . , `z`) then
  - punctuation (`{`, …, `~`)
- See, for example:
  - https://wpollock.com/Docs/ascii.htm
  - http://www.unicode.org/charts/PDF/U0000.pdf
- This controls character comparisons and string ordering using `compareTo`

- It is sometimes useful to convert all the characters of a string to upper case (e.g. if "JAVA" is preferred to "Java" or "java"):

```
String text = "Newspaper Headline";
String headline = text.toUpperCase();
```

  - gives `headline` the value

```
"NEWSPAPER HEADLINE"
```

- There is a comparable `toLowerCase` method
- This is often useful:

```
text = text.toLowerCase();    // "convert"
```

- Since `indexOf` does not have an `IgnoreCase` version, this is useful:

```
String text = someText.toLowerCase();
String s = seekText.toLowerCase();
... text.indexOf(s) ...
```


Strings as parameters

- Like any other object, strings can be provided as method parameters:

```
// Return (shortened) string (at most max)
private String shorten(String s, int max) {
  int len = s.length();  // Get string length
  if (len <= max)        // Within maximum?
    return s;            // Yes, return given
  else
    return s.substring(0, max);
                         // No, return shortened
}
```

Strings – more methods

- There are many more operations on strings; see JFS and the Java documentation
- We sometimes want to replace one character by another throughout a string
  - For example, this code "cleans up" a number represented as a string containing commas:

```
String numberString = "1,000,000,000";
// Replace commas with empty string
numberString = numberString.replace("," , "");
// Now can convert
int number = Integer.parseInt(numberString);
```


- It is also be useful to be able to split a string into an array of substrings separated by some designated character:

```
String data = "Hello World, how are you?";
// split into substrings separated by a space
String[] values = data.split(" ");
```

  - The `values` array will contain the strings

```
"Hello" "World," "how" "are" "you?"
```

- `split` is far more powerful than it appears
  - For example

```
String[] values = data.split("[ ,;]");
```

  - Would split at every space and , and ;
- Note: JFS uses `StringTokenizer` instead of `split`.

End of section