**Strings**

**SYNTACTIC SUGAR AND MUTABILITY**

The statement:

```java
String name = new String("York");
```

is the same as:

```java
String name = "York";
```

To join two strings, use the **coercive +** operator:

```java
String fullName = name + " University";
```

in place of the messy:

```java
String fullName = ((new StringBuffer(name)).append(" University")).toString();
```

The statement:

```java
name.toUpperCase();
```

does not change name. To do so, re-assign name.

**THE STRING API**

**Surgery**

- length
- charAt
- substring

---

**Transforms**

- toLowerCase
- toUpperCase
- trim

---

**Comparators**

- equals
- compareTo
- Empty vs null

---

**Patterns**

- indexOf
- replaceAll
- split / Tokenizer

---

**Regex**

- Pattern compile
- Matcher matcher
- find and group

---

**ABOUT STRINGS**

- They are objects like the rest
  
  *But with syntactic sugar to make instantiation easier and with an operator to make concatenation seem natural.*

- A String is a sequence of chars
  
  *This sequence is the state held by the object.*

- They have an extremely rich API
  
  *Available through a number of classes such as String, StringBuffer, Matcher, Pattern, …*

- They Are Immutable
  
  *For performance purposes.*
In a class with a `String` attribute implement these features:

- A constructor that takes a `String` parameter `Must set the attribute accordingly.`
- `public String get()` and `public void set(String s)` `Accessor and mutator.`
- `public int repeatCount(char c)` and `public String toDayName(int d)` `Returns the number of times c occurs in the state. Returns the state with any 0-6 digit replaced with Sun…Sat.`
- `public String trimLeadingBlanks()` `Returns the state with any leading spaces removed.`

Find the substring `m4n7r2` in this string:

Mine Canadian postal codes in this string:

```
String query = "m4n7r2";
System.out.println(s.indexOf(query));
```

Given the string s:

```
String regex = \[A-Za-z]\d[A-Za-z]\d\";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(s);
while (matcher.find()) {
  // use matcher.group(); .start(); .end();
}
```

### Exercises

**In a class with a `String` attribute implement these features:**

- A constructor that takes a `String` parameter `Must set the attribute accordingly.`
- `public String get()` and `public void set(String s)` `Accessor and mutator.`
- `public int repeatCount(char c)` and `public String toDayName(int d)` `Returns the number of times c occurs in the state. Returns the state with any 0-6 digit replaced with Sun…Sat.`
- `public String trimLeadingBlanks()` `Returns the state with any leading spaces removed.`

### Search vs Pattern Search

**Find the substring `m4n7r2` in this string:**

Mine Canadian postal codes in this string:

```
String query = "m4n7r2";
System.out.println(s.indexOf(query));
```

**Given the string s:**

```
String regex = \[A-Za-z]\d[A-Za-z]\d\";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(s);
while (matcher.find()) {
  // use matcher.group(); .start(); .end();
}
```
Given a String $s$, write a fragment that:

- Mine all telephone numbers in it.
- Determines if $s$ has a person's height (expressed in feet and inches, as in 5'9") imbedded in it.
- Determines the largest word in $s$. We define a word as a sequence of non-space characters. Use $\text{split}$. 
- Validates that $s$ represents a query string. These are used in web applications. They begin with $?$ and consist of one or more $\text{var}=\text{value}$ clauses delimited by $\&$.
- Validates that $s$ is a DNA strand (consists of $A,C,G,T$, begins with ATG and its length is divisible by 3). If so, output the most frequent amino acid in it (any sequence of 3 letters).