Basic Computation
Part 02
• Class types group together data with functionality (methods)
• Classes create instances of Objects
• Separated by Reference and Contents
  – Reference is the memory address that “points” to the object’s contents in memory
  – A reference is the value stored in the identifier
  – Contents contain the data and functionality

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Contents</th>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>objectId</td>
<td>28</td>
<td>14</td>
</tr>
<tr>
<td>..</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>objectId.data01</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>objectId.data02</td>
<td>3.0</td>
<td>32</td>
</tr>
<tr>
<td>objectId.method01()</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>objectId.method02()</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
• Objects must be *constructed* before used
  – Default value for class types is NULL
  – NULL means “nothing” as the object does not exist
  – Cannot use a NULL object
  – Reserved word “new” is used to construct instances of most class types, but not usually for Strings

• Methods provide functionality for an object
  – It’s what the object can do
  – Reusing code

• Methods are *called* by using the object’s identifier, followed by a dot “.”, followed by the method name an arguments

**Syntax for Calling a Method**

```
<<identifier>>.<<method name>>(<<arguments>>);
```
### Class type
- Data = Array of Characters
- Methods = Built-in Functionality

### Denoted by double quotes (""")
- Single Characters are single quotes ('')

### Used to group together single characters into words and phrases
- Useful for Outputting and Formatting Data
- Useful for Inputting Data as words or sentences

### Syntax
```plaintext
String <<identifier>>; // Declare a String
// Assigning a String Value
<<identifier>> = "<<String Value>>";
```
Array of Characters
  - Contiguous Collection of Characters
  - Individual Characters can be accessed by an “index”
  - Indices always start from 0 to Length - 1

Example String

String str = “abcdefg”;

String as an Array

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>‘a’</td>
<td>‘b’</td>
<td>‘c’</td>
<td>‘d’</td>
<td>‘e’</td>
<td>‘f’</td>
<td>‘g’</td>
</tr>
</tbody>
</table>
The plus (+) operator concatenates a value with a String
  – Not the same as the mathematical “+”

Useful methods
  – length()
  – charAt(index)
  – substring(startIndex)
  – substring(startIndex, endIndex)
  – toUpperCase()
  – toLowerCase()
  – split(regular expression)

Examples
String str = “abcdefg”;
System.out.println(str.charAt(0));
String str2 = str.substring(2,5);
System.out.println(str2);
FIGURE 2.5 Some Methods in the Class String

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Type</th>
<th>Example for String s = &quot;Java&quot;;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>charAt (index)</td>
<td>char</td>
<td>c = s.charAt(2);</td>
<td>Returns the character at index in the string. Index numbers begin at 0.</td>
</tr>
<tr>
<td>compareTo (a_string)</td>
<td>int</td>
<td>i = s.compareTo(&quot;C++&quot;);</td>
<td>Compares this string with a_string to see which comes first in lexicographic (alphabetic, with upper before lower case) ordering. Returns a negative integer if this string is first, zero if the two strings are equal, and a positive integer if a_string is first.</td>
</tr>
<tr>
<td>concat (a_string)</td>
<td>String</td>
<td>s2 = s.concat(&quot;rocks&quot;);</td>
<td>Returns a new string with this string concatenated with a_string. You can use the + operator instead.</td>
</tr>
<tr>
<td>equals (a_string)</td>
<td>boolean</td>
<td>b = s.equals(&quot;Java&quot;);</td>
<td>Returns true if this string and a_string are equal. Otherwise returns false.</td>
</tr>
<tr>
<td>equalsIgnoreCase (a_string)</td>
<td>boolean</td>
<td>b = s.equalsIgnoreCase(&quot;Java&quot;);</td>
<td>Returns true if this string and a_string are equal, considering upper and lower case versions of a letter to be the same. Otherwise returns false.</td>
</tr>
<tr>
<td>indexOf (a_string)</td>
<td>int</td>
<td>i = s.indexOf(&quot;va&quot;);</td>
<td>Returns the index of the first occurrence of the substring a_string within this string or -1 if a_string is not found. Index numbers begin at 0.</td>
</tr>
<tr>
<td>Method</td>
<td>Return Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>lastIndexOf(a_string)</code></td>
<td>int</td>
<td>Returns the index of the last occurrence of the substring <code>a_string</code> within this string or -1 if <code>a_string</code> is not found. Index numbers begin at 0.</td>
<td></td>
</tr>
<tr>
<td><code>length()</code></td>
<td>int</td>
<td>Returns the length of this string.</td>
<td></td>
</tr>
<tr>
<td><code>toLowerCase()</code></td>
<td>String</td>
<td>Returns a new string having the same characters as this string, but with any uppercase letters converted to lowercase. This string is unchanged.</td>
<td></td>
</tr>
<tr>
<td><code>toUpperCase()</code></td>
<td>String</td>
<td>Returns a new string having the same characters as this string, but with any lowercase letters converted to uppercase. This string is unchanged.</td>
<td></td>
</tr>
<tr>
<td><code>replace(oldchar,newchar)</code></td>
<td>String</td>
<td>Returns a new string having the same characters as this string, but with each occurrence of <code>oldchar</code> replaced by <code>newchar</code>.</td>
<td></td>
</tr>
<tr>
<td><code>substring(start)</code></td>
<td>String</td>
<td>Returns a new string having the same characters as the substring that begins at index <code>start</code> through to the end of the string. Index numbers begin at 0.</td>
<td></td>
</tr>
<tr>
<td><code>substring(start,end)</code></td>
<td>String</td>
<td>Returns a new string having the same characters as the substring that begins at index <code>start</code> through to but not including the character at index <code>end</code>. Index numbers begin at 0.</td>
<td></td>
</tr>
<tr>
<td><code>trim()</code></td>
<td>String</td>
<td>Returns a new string having the same characters as this string, but with leading and trailing whitespace removed.</td>
<td></td>
</tr>
</tbody>
</table>
Strings

- Object type
- Array of characters

Examples
String str = “abcd”;
Strings

- Object type
- Array of characters

Examples
String str = "abcd";
Strings

- Object type
- Array of characters

Examples

```java
String str = "abcd";
```
Strings

Object type
Array of characters

Examples

String `str = “abcd”;

Memory

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Contents</th>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>str</td>
<td>Null</td>
<td>28</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>str[0]</td>
<td>‘\u0000’</td>
<td>64</td>
</tr>
<tr>
<td>str[1]</td>
<td>‘\u0000’</td>
<td>66</td>
</tr>
<tr>
<td>str[2]</td>
<td>‘\u0000’</td>
<td>68</td>
</tr>
<tr>
<td>str[3]</td>
<td>‘\u0000’</td>
<td>70</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Object type
Array of characters

Examples

String

```java
String str = "abcd";
```
Object type
Array of characters

Examples

String
```
str = "abcd";
```
Strings

- Object type
- Array of characters

**Examples**

String `str = “abcd”;`

**Memory**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Contents</th>
<th>Byte Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>str</td>
<td>64</td>
<td>28</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>str[0]</td>
<td>‘a’</td>
<td>64</td>
</tr>
<tr>
<td>str[1]</td>
<td>‘b’</td>
<td>66</td>
</tr>
<tr>
<td>str[2]</td>
<td>‘c’</td>
<td>68</td>
</tr>
<tr>
<td>str[3]</td>
<td>‘d’</td>
<td>70</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
- Used to better format Strings
- Considered Single Characters
  - Despite there are two individual characters
- Starts with a “\"
- \" - Double Quote
- ‘ - Single Quote
- \\ - Backslash
- \n – New Line. Go to beginning of Next line
- \r – Carriage Return. Go to beginning of the Current line
- \t – Tab. Add space until next tab stop

### Examples

```java
String str = "Hello\n"World\"";
System.out.println(str);
```

### Console

```
Hello
“World”
```
### Class Type
- Used to “Scan” or “Read”
  - Standard System Input “System.in” (Console)
  - Strings
  - Files
  - Network Traffic

### Must import type Scanner from “java.util” package
- import java.util.Scanner;

### Before using it must be both Declared and Constructed
- The “ARGS” part is the item the Scanner will process. It can be the System input, Strings, Files, etc.

### Syntax
```java
//Declaring and Constructing a Scanner
Scanner <identifier> = new Scanner(<ARGS>);
```

### Example
```java
//Declaring and constructing a Scanner for Console (System.in)
Scanner keyboard = new Scanner(System.in);
```
Once a Scanner has been declared and constructed it can be used by calling its various methods

Scanner uses delimiters
- Separates information by Special Characters
- Assumed to be any kind of space unless otherwise declared
- Types of spaces include
  - Single Spaces
  - Multiple Spaces
  - End Line / Carriage Returns
  - Tabs

Examples
Scanner keyboard = new Scanner(System.in);
String name = keyboard.nextLine();
int i = keyboard.nextInt();
keyboard.nextLine();//Useful “fix-up”
double j = keyboard.nextDouble();
keyboard.nextLine();//Useful “fix-up”
System.out.println(name + " " + i + " " + j);

Console
JJ
64
3.14
JJ 64 3.14
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| next()           | Returns a String value up to but not including the first delimiter character | //Assume user enters “1234 3.14 true asdf”  
|                  |                                                                             | String str = keyboard.next();                                            |
|                  |                                                                             | //str is “1234”                                                          |
| nextLine()       | Returns a String value up to but not including the line terminator ‘\n’     | //Assume user enters “1234 3.14 true asdf”  
|                  |                                                                             | String str = keyboard.nextLine();                                       |
|                  |                                                                             | //str is “1234 3.14 true asdf”                                           |
| nextInt()        | Returns the first instance of an integer value. All other characters and     | //Assume user enters “1234 3.14 true asdf”  
|                  | delimiters are ignored.                                                      | int i = keyboard.nextInt();                                              |
|                  |                                                                             | //int i is 1234                                                          |
| nextDouble()     | Returns the first instance of a double value. All other characters and      | //Assume user enters “1234 3.14 true asdf”  
|                  | delimiters are ignored.                                                      | double j = keyboard.nextDouble();                                        |
|                  |                                                                             | //double j is 3.14                                                       |
| nextBoolean()    | Returns the first instance of a Boolean value. All other characters and     | //Assume user enters “1234 3.14 true asdf”  
|                  | delimiters are ignored.                                                      | boolean b = keyboard.nextBoolean();                                      |
|                  |                                                                             | //Boolean b is true                                                      |
Classes that “Wrap” or provide functionality to primitive types

- Can be used to convert a String into a primitive type

- Commonly Used
  - `Integer.parseInt(<String>)`;
  - `Double.parseDouble(<String>)`;
  - `Boolean.parseBoolean(<String>)`;

Examples

```java
String str = "256";
int i = Integer.parseInt(str);
i *= 2;
System.out.println(i);
```

Console

```
512
```
Example in more detail

input =

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>.</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current Line of Code

```java
String input = keyboard.nextLine();
```
Example in more detail

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>.</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
input = copyInput =
```

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>.</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current Line of Code

```java
String copyInput = input;
```
Example in more detail

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>2</td>
<td>.</td>
<td>2</td>
</tr>
</tbody>
</table>

**input =**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>2</td>
<td>.</td>
<td>2</td>
</tr>
</tbody>
</table>

**copyInput =**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>2</td>
<td>.</td>
<td>2</td>
</tr>
</tbody>
</table>

**workingIndex = 3**

Current Line of Code

```java
int workingIndex = copyInput.indexOf(" ");
```
Example in more detail

```
String name = copyInput.substring(0,workingIndex);
```
Example in more detail

Current Line of Code

```java
   copyInput = copyInput.substring(workingIndex+1);
```
Example in more detail

input =

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
<td>2</td>
<td>3</td>
<td></td>
<td>2</td>
<td>.</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

copyInput =

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td></td>
<td>2</td>
<td>.</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

workingIndex = 3

Current Line of Code

copyInput = copyInput.substring(workingIndex+1);
• Documentation and Style is important
  – Most programs are modified over time to respond to new requirements
  – Programs that are easy to read and understand are easy to modify
  – You have to be able to read it in order to debug it
• Meaningful Identifiers
  – Identifiers should suggest its use
  – Stick to common conventions

• Commenting
  – Self documenting with Clean Style is best
  – Comments are written as needed
  – Used by programmers to explain code, but ignored by the compiler
  – Include your name at the beginning of every file
  – It’s good to write an explanatory comment at the beginning of the file
• Indentation
  – Use indentation to “line-up” code within their respective bodies
  – Clearly indicates “nested” statements