1 Strings, Characters and Regular Expressions

CST242

2 char and String Variables

- A char is a Java data type (a primitive numeric) that uses two bytes (16 bits) to store one text character …
  - char literals enclosed in single quotes
  - E.g. char anyLetter = 'L';
- A String (object or reference) is a series of characters treated as a unit …
  - String literals enclosed in double quotes
  - E.g. String firstName = "Charles";

3 Character Representation

- All characters (whether in a char or a String) are represented as a binary integer value between zero (0) and 65,535
- Requires two bytes (16 bits) of storage in RAM or on a disk …
  - The highest *16 digit* binary number is 11111111 11111111 or 65,535
  - Written in hexadecimal as FFFF
- The integer storage values are known as Unicode (formerly ANSI—which was one byte)

4 The Unicode Table

- Complete Unicode specification can be found at:
  - http://www.ssec.wisc.edu/~tomw/java/unicode.html
  - The letter "A" is:
    - 65 in decimal
    - 0000 0000 0100 0001 in Unicode binary (0041 in hexadecimal)
  - The letter "a" is:
    - 97 in decimal
    - 0000 0000 0110 0001 in Unicode binary (0061 in hexadecimal)

5 The StringBuilder Class (Page 1)

- A class that provides functionality for building and concatenating strings into a single string
- StringBuilder class is located in the java.lang package (does *not* need to be imported)
- Documentation located at:
  - https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

6 The StringBuilder Class (Page 2)

- The primary methods of class StringBuilder are:
  - append—concatenates String (or some other type since the method is overloaded and converts it to String) *to the end* of the StringBuilder object
  - insert—inserts String (or some other type converted to String) *within* the StringBuilder object

  - delete—deletes the specified elements from a location within the StringBuilder object

7 □ Instantiating StringBuilder Objects (Page 1)

- There are four constructors including:

  StringBuilder *stringBuilderObject* = new StringBuilder();

  - Creates a StringBuilder object with a capacity of 16 elements (initially empty)

  StringBuilder *stringBuilderObject* = new StringBuilder(*initialCapacity*);

  - Creates empty StringBuilder with capacity specified by parameter *initialCapacity*

8 □ Instantiating StringBuilder Objects (Page 2)

- There are four constructors including (*con.*):

  StringBuilder *stringBuilderObject* = new StringBuilder(*object*);

  - Creates a StringBuilder with the initial value of the specified *object* plus 16 additional empty elements

9 □ The append Method (Page 1)

- StringBuilder method that concatenates its argument to the end of StringBuilder object
- The data is converted to a String before the append operation takes place
  - Therefore the argument *type* may be String or any of the following:
    - boolean, char, char[] (array), float, double, short, int, long, Object, etc.

10 □ The append Method (Page 2)

- Format:

  stringBuilderObject.append(*object*);

  - *object* is the element added to the end of the StringBuilder object
- Example:

  output.append("The char is ");

  output.append(c1);

11 □ The insert Method (Page 1)

- StringBuilder method of that inserts the second argument into StringBuilder object
- The first int argument indicates the index before which the data is to be inserted
- Like append(), the data is converted to a string before the insert operation takes place
  - Therefore the argument *type* may be String, but also may be boolean, char, char[] (i.e a char array), float, double, short, int, long, or Object

12 □ The insert Method (Page 2)

- Format:

  stringBuilderObject.insert(*index*, *argument*);

  - *index* is the location before which the *argument* value is to be inserted
- Example:

  output.insert(10, s2);

13 The delete Method

- StringBuilder method that deletes subsequence of characters from *start* to *end* (exclusive) in the StringBuilder object
- Format:

  stringBuilderObject.delete(*start*, *end*);
- Example:

  output.delete(0, output.length());
  - This example deletes all characters from the StringBuilder object

14 The length Method

- Like the String class, class StringBuilder has a length method
- Returns an int as the number of characters in the string builder
- Format:

  stringBuilderObject.length()

15 The capacity Method

- The capacity, which is an int returned by the capacity method, is always greater than or equal to the length
- Automatically expands as necessary to accommodate additions to the string builder
- Format:

  stringBuilderObject.capacity()

16 The deleteCharAt Method

- StringBuilder method that deletes the char located at the *index* in the string builder object
- Format:

  stringBuilderObject.deleteCharAt(*index*);
- Example:

  output.deleteCharAt(0);
  - This deletes the 1st char of the string builder object

17 The replace Method (Page 1)

- StringBuilder method that replaces the specified characters in a string builder object
- Format:

  stringBuilderObject.replace(*start*, *end*, *stringObject*);

18 The replace Method (Page 2)

- Example:

  output.replace(2, 4, "Hello");
  - This example replaces the 3rd through the 4th char's of the string builder object with the string "Hello"

19 The reverse Method

- StringBuilder method that reverses sequence of characters in the string builder object

- Format:

  stringBuilderObject.reverse();

20 □ The setCharAt Method

- StringBuilder method that replaces a single character in the string builder object
- Format:

  stringBuilderObject.setCharAt(*index*, *char*);
- Example:

  output.setCharAt(8, 'G');
  - This example replaces the 9th char of the string builder object with the character 'G'

21 □ The toString Method (Page 1)

- StringBuilder has a toString() method that overrides that of Object and returns a string representation of the object
  - Effectively the character sequence within the StringBuilder object

22 □ The toString Method (Page 2)

- Format:

  stringBuilderObject.toString()
- Examples:

  String s2 = output.toString();
  - Return type of method is String

  System.out.println(output.toString());

  JOptionPane.showMessageDialog(null, output.toString());

24 □ The String Class (Page 1)

- String variables are reference variables (objects of class String) …
  - Represent *multiple* locations in RAM (the characters plus its methods)
- The String class is located in the java.lang package, so it does *not* need to be imported

25 □ The String Class (Page 2)

- String objects contain methods used for manipulating them …
  - Java methods for processing strings include techniques for finding/comparing characters, extracting substrings, modifying upper/lower case, etc.
- Documentation located at:
  - https://docs.oracle.com/javase/9/docs/api/java/lang/String.html

26 □ Instantiating Strings (Page 1)

- Java Strings may be declared using the same format as primitive variables (declares an un-instantiated String object):
  - Format:

    String *variableName*;
- Or a string may be instantiated formally using *object-oriented* notation with a constructor call:
  - Format:

String *variableName* = new String();

27 □ **Instantiating Strings (Page 2)**

- There are 11 *constructor* methods for instantiating String objects
- Example with no arguments:

  String middleName = new String();

28 □ **Instantiating Strings (Page 3)**

- Example with String arguments:

  String lastName = new String("Jenson");

  - Equivalent to: String lastName = "Jenson";
- Example with String variable argument (actually the *same constructor* as above):

  String lastName = new String(s1);

  - Equivalent to: String lastName = s1;

29 □ **Instantiating Strings (Page 4)**

- Other String constructors accept char arrays, byte arrays, StringBuffers and StringBuilders

31 □ **Methods of the String Class**

- Used to perform manipulations with or upon the String object
- Formats:

  stringVariable.*method*([*arg1*, *arg2*, …])

  "*string*".*method*([*arg1*, *arg2*, …])
- Some examples:

  int stringLength = s1.length();

  if (s1.equals("Java")) {…}

  int indexLocation = "hello".indexOf(s5);

  String subStr1 = s1.substring(12);

32 □ **The length Method**

- A method of the String class that returns an int which is the count of the *number of characters* within a String object
- Format:

  stringObject.length()
- Examples:

  int stringLength = s1.length();

  int stringLength = "hello".length();

  - The second example returns the integer 5

33 □ **The charAt Method (Page 1)**

- Method of class String that returns a char (*one character*) from a specific location within the String object and converts it to a char
- Format:

stringObject.charAt(*index*)

- *index* is an integer (its position) within *stringObject* starting at zero (0) to one less than its length

34 □ The charAt Method (Page 2)

- Examples:

  char letter = s1.charAt(7);

  char letter = "hello".charAt(1);

  - The second example returns the character 'e'

36 □ The equals and equalsIgnoreCase Methods (Page 1)

- A boolean method of class String that compares its String object to another String to see if they are identical
  - Returns a value of true or false
- The equals method is contained in Object class and *inherited* by the String class …
  - *Overrides* the same method of its superclass Object

37 □ The equals and equalsIgnoreCase Methods (Page 2)

- The equalsIgnoreCase method ignores the *upper/lower case* of the letters compared …
  - Internally in the ALU, the processor changes the *11th* Unicode position to a 1 if necessary
  - Example:
    - "H" in binary: 00000000 01001000
    - "h" in binary: 00000000 01101000

38 □ The equals and equalsIgnoreCase Methods (Page 3)

- Formats:

  stringObject.equals(*String*)

  stringObject.equalsIgnoreCase(*String*)

  - The *String* argument may be a String variable or String literal to which the *stringObject* is compared
- Examples:

  if (s1.equals("Java")) …

  - Equivalent but *invalid*: if (s1 == "Java") …

  if (s2.equalsIgnoreCase(s3)) …

39 □ The equals and equalsIgnoreCase Methods (Page 4)

- Why is it not possible to use "is equal to" operator (==) with Strings?
- String is a class and so Strings are *objects*
- When used with two objects "is equal to" operator asks if the two objects are identical, that is do they share same address in memory
- The following (compares addresses) really means "are these two objects the same String?":

  if (s1 == "Java")

40 String Comparison Processing

- Made character by character, from *left* to *right*, in accordance with the computer's collating sequence
  - Unicode (ANSI, ASCII) , EBCDIC or some other code
- The binary value of the leftmost character of *one factor* is compared to the binary value of the leftmost character of *the other*
- If they are equal, the comparisons continue with each succeeding character position

41 String Comparison Examples

- Example 1:
  - "java"
    - Binary: 0110 1010 (106) / 0110 0001 (97) …
  - "jello"
    - Binary: 0110 1010 (106) / 0110 0101 (101) …
- Example 2:
  - "hello"
    - Binary: 0110 1000 (104) / 0110 0101 (101) …
  - "Hello"
    - Binary: 0100 1000 (72) / 0110 0101 (101) …

43 The compareTo and compareToIgnoreCase Methods (Page 1)

- Methods of String class that compare the String object to another String to see if the object is:
  - *Greater* or *lesser* than the String argument to which it is compared
  - *Equal* to the String argument to which it is compared (alternative to equals and equalsIgnoreCase)

44 The compareTo and compareToIgnoreCase Methods (Page 2)

- The return values is an int as follows:
  - A *positive* integer if the String object is greater than the “compare to” String argument
  - A *negative* integer if the String object is less than the “compare to” String argument
  - *Zero* (0) if the String object is equal to the “compare to” String argument
- The compareToIgnoreCase method ignores the *case* of the letters compared

45 The compareTo and compareToIgnoreCase Methods (Page 3)

- Formats:

  stringObject.compareTo(*String*)

  stringObject.compareToIgnoreCase(*String*)

  - The *String* may be a String variable or String literal to which the *stringObject* is compared
- Examples:

  ```
  if ( s1.compareTo("Java") > 0) {…}
  ```

- Equivalent but *invalid*: if (s1 > "Java")

  if (s2.compareToIgnoreCase(s3) < 0) {…}

47 □ The regionMatches Method (Page 1)

- A boolean method of the String class that compares *portions* of two strings to determine if they are identical
  - Returns a value of true or false
- Arguments specify where in the strings the comparison begins and for how many consecutive characters

48 □ The regionMatches Method (Page 2)

- Format 1:

  stringObject.regionMatches(*startIndex*, *compareString, startIndexCompareString, numberOfChars*)
  - *startIndex* is the starting location in the *stringObject*
  - The *compareString* may be a String variable or String literal to which the *stringObject* is compared
  - *startIndexCompareString* is location in compare String argument where the comparison begins
  - *numberOfChars* is number of characters to compare

49 □ The regionMatches Method (Page 3)

- Format 2:

  stringObject.regionMatches(true|false, *startIndex*, *compareString, startIndexCompareString, numberOfChars*)
  - If true|false literal is specified as the first argument, comparison is case insensitive
    - If the value is true, comparison is case insensitive
  - All other arguments are *identical* to Format 1
  - In both formats, if String is shorter than *numberOfChars* of characters to be returned, reads addition garbage characters in RAM beyond the String object

50 □ The regionMatches Method (Page 4)

- Examples:

  if (s1.regionMatches(2, s2, 2, 5)) …

  if (s3.regionMatches(true, 2, s4, 2, 5)) …

52 □ The indexOf Method (Page 1)

- Returns an int which is index (zero-based integer) of *first* location of a char or String within string object
  - Returns -1 if the char or String is *not* found
- Format:

  stringObject.indexOf(*char*|*String* [, *index*])
  - First argument is *character*(*s*) searched for
  - Optional *index* argument is starting location for search

    - Or begins at *start* of String

53 □ **The indexOf Method** **(Page 2)**

- Examples:

```
int indexLocation = s1.indexOf(s2);
int indexLocation = s3.indexOf('c');
int indexLocation = s4.indexOf("hello");
int indexLocation = s5.indexOf(s6, 12);
```

54 □ **The lastIndexOf Method** **(Page 1)**

- Returns an int which is the index value of *last* location of char or String substring within string object
  - Returns -1 if the char or String is *not* found
- Format:

  stringObject.lastIndexOf(*char* | *String* [, *index*])
  - First argument is the *character*(*s*) searched for
  - *Optional* index argument is the starting location for the search (searches *before* index) or search starts at *end* of String

55 □ **The lastIndexOf Method** **(Page 2)**

- Examples:

```
int lastIndexLocation = s1.lastIndexOf(s2);
int lastIndexLocation = s3.lastIndexOf('c');
int lastIndexLocation = s4.lastIndexOf("hello");
int lastIndexLocation = s5.lastIndexOf(s6, 12);
```

57 □ **The substring Method** **(Page 1)**

- Returns a String which is the subset of characters from within a string beginning at specified *start* location
  - If an optional *stop* location is designated, characters are returned only up to that location
  - Otherwise, *all* characters to the end of the string object are returned
- Although the characters are returned, the *original* String object is *unchanged*

58 □ **The substring Method** **(Page 2)**

- Format:

  stringObject.substring(*startIndex*[, *stopIndex*])
  - *startIndex* is an int which is the location in *stringObject* where copying of characters *begins*
  - *stopIndex* is an int which is the location in *stringObject* where the subset of characters returned *stops*
    - *Optional* argument meaning exclusive, only characters up to but not including it

59 □ **The substring Method** **(Page 3)**

- Examples:

String s2 = s1.substring(12);

– Returns all characters from index position 12 to end of string (the 13th character)

String s4 = s3.substring(12, 16);

– Returns all characters from index position 12 up to *but not including* index position 16 (the 17th character)

61 The concat Method (Page 1)

- Returns a String which is the concatenation of String argument to the end of String object
- Used optionally in place of concatenation (+) operator

62 The concat Method (Page 2)

- Format:

  stringObject.concat(*String*)

  - The *String* argument (String variable or String literal) is the value concatenated to the *stringObject*
- Example:

  String s3 = s1.concat(s2);

  - If *s1* = "hello" and *s2* = "goodbye" ...
  - The concatenated String in *s3* = "hellogoodbye"

64 The toLowerCase Method (Page 1)

- Returns a String with all the alphabetic characters in the String object converted to *lower case* ...
  - Adds binary 1 to *11th* bit from left
  - Effects *only* alphabetic characters
- Although the lowercase characters are returned, the original String object is *unchanged*

65 The toLowerCase Method (Page 2)

- Format:

  stringObject.toLowerCase()

  - There are *no arguments* to the method
- Examples:

  String s2 = s1.toLowerCase();

  - If s1 = "Hello" ... the String s2 = "hello"

  s1 = s1.toLowerCase();

  - The variable s1 is *updated* to store "hello"

66 The toUpperCase Method (Page 1)

- Returns a String with all the alphabetic characters in the String object converted to *upper case*
  - Subtracts binary 1 from *11th* bit from left
  - Effects *only* alphabetic characters

- Although the uppercase characters are returned, the original String object is *unchanged*

67 The toUpperCase Method (Page 2)

- Format:
  - *stringObject*.toUpperCase()
  - There are *no arguments* to the method
- Example:
  - String s2 = s1.toUpperCase();
  - If s1 = "Hello" ... the string s2 = "HELLO"
  - s1 = s1.toUpperCase();
  - The variable s1 is *updated* to store "HELLO"

69 The replace Method (Page 1)

- Returns a String with *all instances* of one specific character within the String object *replaced* by specified char or char *variable*
- Although the String with the characters replaced is returned, the original String object is *unchanged*

70 The replace Method (Page 2)

- Format:
  - *stringObject*.replace(*char1/String1*, *char2/String2*)
  - *char1/String1* are the character(s) *being replaced*
  - *char2/String2* are the character(s) *replacing the first character(s)*
- Example:
  - String s2 = s1.replace('l', 'Z');
  - If s1 = "Hello" ... the string s2 = "HeZZo"
  - s1 = s1.replace('l', 'Z');
  - The variable s1 is *updated* to store "HeZZo"

71 The trim Method (Page 1)

- Returns a String with all the *leading* and *trailing* blank spaces *stripped* from the String object
- Although a String with the blanks removed is returned, the original String object is *unchanged*

72 The trim Method (Page 2)

- Format:
  - *stringObject*.trim()
  - There are *no arguments* to the method
- Example:
  - String s2 = s1.trim();
  - If s1 = " hello goodbye " ... the string s2 = "hello goodbye"

93 The split Method (Page 1)

- Splits a String object into tokens
  - Tokens are a series of substrings or a collection of String objects (like an array)
- For example:
  - In the String:
    - "Tokens are sets of characters"
  - The tokens are:
    - "Tokens", "are", "sets", "of", "characters"
  - Assuming the blank space (" ") is the delimiter

94 □ The split Method (Page 2)

- Format:

  stringObject.split(*regExpession*)

  - *regExpression* is a regular expression, e.g. the String which is the delimiter between the tokens
- Example:

```
String[] t1 = s1.split(" ");
```

98 □ The toString Method (Page 1)

- Remember the toString method is a member of the class Object from which all classes extend …
  - All classes inherit toString from class Object (or through the superclass of the class) and may call the method directly if not overridden
- Method toString of class String *overrides* method from class Object
- Returns the *string value contents*

99 □ The toString Method (Page 2)

- Formats:

  stringObject.toString()
- Examples:

```
JOptionPane.showMessageDialog(null,  s1.toString() );
JOptionPane.showMessageDialog(null,  s1);
JOptionPane.showMessageDialog(null,  "hello".toString() );
```

100 □ The toString Method (Page 3)

- So what is the difference between:

```
JOptionPane.showMessageDialog(null,
   firstName + " " + lastName);
```

- And:

```
JOptionPane.showMessageDialog(null,
   firstName.toString() + " ".toString()
   + lastName.toString() );
```

- *None*—both call the toString methods of their String objects

101 □ The Character Class (Page 1)

- Character is a "wrapper" class that allows primitive char variables to be treated as objects
- Located in the java.lang package (does *not* need to be imported)
- Documentation located at:
  - http://download.oracle.com/javase/9/docs/api/java/lang/Character.html

102 □ The Character Class (Page 2)

- There is a single constructor for the Character class
- Constructor has been *deprecated* and is marked for removal in a future version of Java
  - Still works in Java 19
- Format:
  - Character *char* = new Character(*char*);
- Example:
  - Character c3 = new Character(c1);
-

103 □ The Character Class (Page 3)

- Most methods are static and take a char argument to either *test* the argument or *manipulate* it in some way
- Format:

  Character.*method*(*char*)
  - No object is instantiated from the Character class
- Examples:

  if (Character.isLetter('c'));

  char c2 = Character.toUpperCase(c1);

104 □ The isDefined Method

- A static boolean method of the Character class that determines if the char argument is defined in *Unicode* character set
  - Returns either true or false
- Format:

  Character.isDefined(*char*)
  - *char* is char literal or char variable being evaluated
- Example:

  if (Character.isDefined(c1)) {…}

105 □ The isDigit Method

- A static boolean method of the Character class that determines if char argument is a digit (0-9)
  - Returns either true or false
- Formats:

  Character.isDigit(*char*)

  - *char* is char literal or char variable being evaluated
- Example:
  if (Character.isDigit(c1)) {…}

106 The isLetter Method

- A static boolean method of the Character class that determines if the char argument is an *alphabetic* character (a-z *or* A-Z)
  - Returns either true or false
- Formats:
  Character.isLetter(*char*)
  - *char* is char literal or char variable being evaluated
- Example:
  if (Character.isLetter(c1)) {…}

107 The isLetterOrDigit Method

- A static boolean method of the Character class that determines if char argument is an *alphabetic* character (a-z *or* A-Z) or digit (0-9)
  - Returns either true or false
- Formats:
  Character.isLetterOrDigit(*char*)
  - *char* is char literal or char variable being evaluated
- Example:
  if (Character.isLetterOrDigit(c1)) {…}

108 The isLowerCase Method

- A static boolean method of the Character class that determines if the char argument is an *lower case* alphabetic character (a-z)
  - Returns either true or false
- Formats:
  Character.isLowerCase(*char*)
  - *char* is char literal or char variable being evaluated
- Example:
  if (Character.isLowerCase(c1)) {…}

109 The isUpperCase Method

- A static boolean method of the Character class that determines if the char argument is an *upper case* alphabetic character (A-Z)
  - Returns either true or false
- Formats:
  Character.isUpperCase(*char*)
  - *char* is char literal or char variable being evaluated
- Example:
  if (Character.isUpperCase(c1)) {…}

110 □ The toLowerCase Method (Page 1)

- A static char method of the Character class that returns an alphabetic char converted to *lower case*
  - Adds binary 1 to 11th bit from left of the char
  - Effects *only* alphabetic characters
- Although a lowercase char is returned, the original char argument is *unchanged*

111 □ The toLowerCase Method (Page 2)

- Format:

  Character.toLowerCase(*char*)
  - *char* is char literal or char variable is the character that is being modified
- Example:

  char c2 = Character.toLowerCase(c1);
  - If c1 = 'C' … then char c2 = 'c'

112 □ The toUpperCase Method (Page 1)

- A static char method of the Character class that returns an alphabetic char converted to *upper case*
  - Subtracts binary 1 from 11th bit of the char
  - Effects *only* alphabetic characters
- Although an uppercase char is returned, the original char argument is *unchanged*

113 □ The toUpperCase Method (Page 2)

- Format:

  Character.toUpperCase(*char*)
  - *char* is char literal or char variable is the character that is being modified
- Example:

  char c2 = Character.toUpperCase(c1);
  - If c1 = 'c' … then char c2 = 'C'

114 □ The charValue Method

- Non-static method charValue returns a char which is the "value" of the character variable or literal
- Format:
  - *char*.charValue()
- Example:
  - System.out.println(c1.charValue());
  - If c1 = 'c' then prints c to the console

115 □ The equals Method

- Non-static method equals returns a boolean value indicating if the value of the char variable or literal is equal to the char argument
- Format:
  - *char*.equals(*char*)

- Example:
  - if (c1.equals(c2)) { … };
  - Equivalent to:
    - if (c1 == c2) { … }

116 □ The compareTo Method

- Non-static method compareTo returns an int value indicating if the value of the char variable or literal is less than or greater than or equal to the char argument
- Format:
  - *char*.compareTo(*char*)
- Example:
  - if (c1.compareTo(c2) > 0) { … };
  - Equivalent to:
    - if (c1 > c2) { … }

118 □ Classes for Manipulation of Other Primitive Types (Page 1)

- There are classes for other *primitive variables* in addition to the Character class
  - Called wrapper classes
- Include the classes Boolean, Double, Float, Byte, Short, Integer and Long, e.g.
  Double.parseDouble
- These classes allow primitive variables to be treated as objects

119 □ Classes for Manipulation of Other Primitive Types (Page 2)

- Examples:

```
Integer grossPay;
ArrayList<Double> payments;
ObservableList<Float> hours;
```

- Classes for primitive variables are located in the java.lang package (do *not* need to be imported)

124 □ Regular Expressions (Page 1)

- A regular expression (regex) is a String pattern that the “regular expression engine” uses to attempt to match input text
- The pattern consists of one or more character literals and/or operators and/or other constructs
- Regular expressions can be used in a wide variety of platforms and languages including Java

125 □ Regular Expressions (Page 2)

- Characters may be one (1) or several characters
- When more than one (1) character, they are placed inside square [brackets]
- A range is specified with a dash (-)
  - [A-Z] means all uppercase characters from A to Z
  - [a-z] means all lowercase characters from a to z

  - [a-zA-Z] means all lowercase and uppercase characters
  - [aeiou] means all lowercase vowels

126 □ Regular Expressions (Page 3)

- Predefined classes offer convenient short-hands for *commonly used* regular expressions:
  - . the dot (.) means any keyable character
  - \d any digit
    - So that "\d{3}" means exactly three digits
  - \w any word character
  - \s any white space character

127 □ Regular Expressions (Page 4)

- Quantifiers indicate (count) how many of the *previous* expression are required for a match:
  - * matches zero (0) or more occurrences
  - + matches one (1) or more occurrences
  - ? matches zero (0) or one (1) occurrence
  - {*n*} matches exactly *n* occurrences
  - {*n*,} matches *n* or more occurrences
  - {*n*,*m*} matches between *n* and *m* occurrences

128 □ The matches Method (Page 1)

- Java uses the boolean method matches which is a member of the String class for implementing the “regular expression engine”
- Tells whether or not a String matches the given regular expression
- Based on the result it returns either true or false

129 □ The matches Method (Page 2)

- Format:

  stringObject.matches("*regEx*");
  - *regex* is the regular expression as a String
- Example:

  if (zipCode.matches("\\d{5}")) ...
  - Matches exactly five digits
  - Since the backslash (\) is a Java “escape” character, it requires two backslashes to represent one backslash, e.g. "\\"

130 □ Characters (Page 1)

- Characters include any typeable (on the computer keyboard) text
- Examples:
  - Starts with one uppercase letter
  - Followed by a combination of zero (0) or more lowercase and/or uppercase letters "[A-Z][a-zA-Z]*"

131 □ Characters (Page 2)

- Examples (*con.*):
  - Starts with one or more either lowercase or uppercase letters
  - Followed in (parentheses) by a combination of zero (0) or more:
    - A single apostrophe (') or dash (-)
    - One or more lowercase and uppercase letters

  "[a-zA-Z]+(['-][a-zA-Z]+)*"

132 □ Characters (Page 3)

- Examples (*con.*):
  - Eight or more of any lowercase and/or uppercase letters

    "[a-zA-Z]{8,}"

133 □ Characters (Page 4)

- Examples (*con.*):
  - An address
    - One or more digits (numeric address)
    - One space
    - In (parentheses)
      - One or more lowercase and/or uppercase letters (the name of the street, avenue, etc.)
      - One or more spaces
      - One or more lowercase and/or uppercase letters (e.g. Street, Avenue, etc.)

  "\\d+\\s+([a-zA-Z]+\\s[a-zA-Z]+)"

134 □ The Dot (.) Wildcard

- The dot (.) is used as a wildcard meaning it represents any character
- Examples:
  - Matches exactly five of any characters

    ".{5}"
  - Matches any eight or more characters

    ".{8,}"

135 □ Phone Numbers (Page 1)

- Simple phone number with dashes:
  - Starts with 1 digit (not zero) and then two digits followed a dash
  - Then another 1 digit (not zero) and then two digits followed by a dash
  - Then four digits
  - E.g. 999-999-9999

    "[1-9]\\d{2}-[1-9]\\{d2}-\\d{4}"

136 □ Phone Numbers (Page 2)

- Phone number which accepts either of two versions:

1. The version with dashes from the previous page; or
2. The version with parentheses, e.g. (999) 999-9999
   `"^\\(?([0-9]{3})\\)?[-.\\s]?([0-9]{3})[-.\\s]?([0-9]{4})$"`

137 □ **Social Security Numbers** **(Page 1)**

- Validating Social Security Numbers (SSN's) may be a bit deceiving and more difficult to validate than might be expected
- The Social Security Administration, on June 25th, 2011, revised their assignment process to use a system of randomization for generating numbers
- Not possible to throw any values in and expect it to be a valid number since there still are a few SSN's that are "off limits"

138 □ **Social Security Numbers** **(Page 2)**

- Simple—a hyphen-separated SSN:
  - The caret (^) and dollar sign ($) represent the beginning and end of the expression
  - Starts with three digits
  - Followed by a dash (-)
  - Followed by two digits
  - Followed by a dash (-)
  - Followed by four digits
    `"^\\d{3}-\\d{2}-\\d{4}$"`

139 □ **Social Security Numbers** **(Page 3)**

- Will accept SSN in the form of 123-45-6789 OR 123456789:
  - The pipe (|) symbol means "or"
    `"^(\\d{3}-\\d{2}-\\d{4})|(\\d{3}\\d{2}\\d{4})$"`

140 □ **Social Security Numbers** **(Page 4)**

- Uses current SSN randomization rules effective since June 25, 2011
  - Validates 9 digit numbers, not separated or separated by dash (-) or space
  - Not starting with 000, 666, or 900-999
  - Not containing 00 or 0000 in the middle or at the end
    `"^(?!000)(?!666)([0-8]\\d{2}) ([ -])? (?!00)\\d\\d ([ -])? (?!0000)\\d{4}$"`

141 □ **E-mail Addresses** **(Page 1)**

- E-mail validation can go from very simple to quite complex
- The simplest e-mail validation:
  `"^(.+)@(.+)$"`

142 □ **E-mail Addresses** **(Page 2)**

- Adding restrictions on the username part:
  - Multiple A-Z and a-z characters allowed
  - Multiple 0-9 numbers allowed
  - Additionally may contain only dot (.), dash (-) and underscore (_) characters

^[A-Za-z0-9+_.-]+@(.+)$

143 □ **E-mail Addresses** **(Page 3)**

- Adding restrictions on the username and the domain parts:
  - One or more words (\w) and dots (.) before the ampersand (@)
  - One or more words (\w) and dots (.) after the ampersand (@)
  - A word of two to four characters after the last dot (.), e.g. ".com", ".uk", etc
    "^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$"
-

144 □ **E-mail Addresses** **(Page 4)**

- Allowing e-mail addresses permitted by RFC 5322 (the organization that governs e-mail address format):
  ^[a-zA-Z0-9_!#$%&'*+/=?`{|}~^.-]+@[a-zA-Z0-9.-]+$

145 □ **Dates** **(Page 1)**

- Dates can be simple without validation, or can validate dates with number of days in a month and validate Feb 29th for leap years
- Dates with:
  - Slashes (/)
  - One or two digit month and date
  - A four digit year
    "^\\d{1,2}/\\d{1,2}/\\d{4}$"

146 □ **Dates** **(Page 2)**

- Dates with:
  - Dashes (-)
  - One or two digit month and date
  - A two digit year:
    "^\\d{1,2}-\\d{1,2}-\\d{2}$"

148 □ **Regex Examples on the Web**

- Learning regular expressions can take a great deal of time and effort
- Many programmers/developers will search a wide number of websites that give many regular expressions for free download and usage
- One of the better ones is:
  - www.regexlib.com