STRING & ARRAY
19TH WEEK LECTURE

엄현상(Eom, Hyeonsang)
School of Computer Science and Engineering
Seoul National University

© COPYRIGHTS 2020 EOM, HYEONSANG ALL RIGHTS RESERVED
Outline

- **String**
  - Overloading
  - Operations
  - Formatter Class
  - Scanner

- **Array**
Immutable Strings

- Objects of the `String` class are immutable.
- `String` class
  - every method in the class that appears to modify a `String` actually creates and returns a brand new `String` object containing the modification.
- The original `String` is left untouched.

```java
import static net.mindview.util.Print.*;
public class Immutable {
    public static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = "howdy";
        print(q); // howdy
        String qq = upcase(q);
        print(qq); // HOWDY
        print(q); // howdy
    }
}
```

>> howdy
HOWDY
howdy
Overloading ‘+’ vs. StringBuilder

• operator ‘+’ has been overloaded for String objects.

• Overloading
  – an operation has been given an extra meaning when used with a particular class.
  – (The ‘+’ and ‘+=’ for String are the only operators that are overloaded in Java, and Java does not allow the programmer to overload any others.)
  – The ‘+’ operator allows you to concatenate Strings

```java
public class Concatenation {
    public static void main(String[] args) {
        String mango = "mango";
        String s = "abc" + mango + "def" + 47;
        System.out.println(s);
    }
}
```

>> abcmangodef47
public class UsingStringBuilder {
    public static Random rand = new Random(47);
    public String toString() {
        StringBuilder result = new StringBuilder("[");
        for(int i = 0; i < 25; i++) {
            result.append(rand.nextInt(100));
            result.append(", ");
        }
        result.delete(result.length()-2, result.length());
        result.append("]");
        return result.toString();
    }
    public static void main(String[] args) {
        UsingStringBuilder usb = new UsingStringBuilder();
        System.out.println(usb);
    }
}
Overloading ‘+’ vs. StringBuilder

- `append(a + " : " + c)`
  - the compiler will jump in and start making more `StringBuilder` objects again.
- `StringBuilder` has a full complement of methods
  - `insert()`, `replace()`, `substring()` and even `reverse()`
- But the ones you will generally use are `append()` and `toString()`.
- Note the use of `delete()` to remove the last comma and space before adding the closing square bracket.
- `StringBuilder` introduced in Java SE5
  - Prior: `StringBuffer`, which ensured thread safety
## Operations on Strings

<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments, Overloading</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td></td>
<td>Number of characters in the String.</td>
</tr>
<tr>
<td>charAt()</td>
<td>int Index</td>
<td>The char at a location in the String.</td>
</tr>
<tr>
<td>getChars(), getBytes()</td>
<td>The beginning and end from which to copy, the array to copy into, an index into the destination array.</td>
<td>Copy chars or bytes into an external array.</td>
</tr>
<tr>
<td>toCharArray()</td>
<td></td>
<td>Produces a char[] containing the characters in the String.</td>
</tr>
<tr>
<td>equals(), equalsIgnoreCase()</td>
<td>A String to compare with.</td>
<td>An equality check on the contents of the two Strings.</td>
</tr>
<tr>
<td>compareTo()</td>
<td>A String to compare with.</td>
<td>Result is negative, zero, or positive depending on the lexicographical ordering of the String and the argument. Uppercase and lowercase are not equal!</td>
</tr>
<tr>
<td>contains()</td>
<td>A CharSequence to search for.</td>
<td>Result is true if the argument is contained in the String.</td>
</tr>
<tr>
<td>Method</td>
<td>Arguments, Overloading</td>
<td>Use</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>contentEquals()</td>
<td>A CharSequence or StringBuffer to compare to.</td>
<td>Result is true if there's an exact match with the argument.</td>
</tr>
<tr>
<td>equalsIgnoreCase()</td>
<td>A String to compare with.</td>
<td>Result is true if the contents are equal, ignoring case.</td>
</tr>
<tr>
<td>regionMatches()</td>
<td>Offset into this String, the other String and its offset and length to compare. Overload adds &quot;ignore case.&quot;</td>
<td>boolean result indicates whether the region matches.</td>
</tr>
<tr>
<td>startsWith()</td>
<td>String that it might start with. Overload adds offset into argument.</td>
<td>boolean result indicates whether the String starts with the argument.</td>
</tr>
<tr>
<td>endsWith()</td>
<td>String that might be a suffix of this String.</td>
<td>boolean result indicates whether the argument is a suffix.</td>
</tr>
<tr>
<td>indexOf(), lastIndexOf()</td>
<td>Overloaded: char, char and starting index, String,</td>
<td>Returns -1 if the argument is not found within this String; otherwise, returns</td>
</tr>
<tr>
<td>Method</td>
<td>Arguments, Overloading</td>
<td>Use</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>substring() (also subSequence())</td>
<td>Overloaded: starting index; starting index + ending index.</td>
<td>Returns a new String object containing the specified character set.</td>
</tr>
<tr>
<td>concat()</td>
<td>The String to concatenate.</td>
<td>Returns a new String object containing the original String's characters followed by the characters in the argument.</td>
</tr>
<tr>
<td>replace()</td>
<td>The old character to search for, the new character to replace it with. Can also replace a CharSequence with a CharSequence.</td>
<td>Returns a new String object with the replacements made. Uses the old String if no match is found.</td>
</tr>
<tr>
<td>toLowerCase() toUpperCase()</td>
<td></td>
<td>Returns a new String object with the case of all letters changed. Uses the old String if no changes need to be made.</td>
</tr>
<tr>
<td>trim()</td>
<td></td>
<td>Returns a new String object with the whitespace removed from each end. Uses the old String if no changes need to be made.</td>
</tr>
<tr>
<td>valueOf()</td>
<td>Overloaded: Object, char[], char[] and offset and count, boolean, char, int, long, float, double.</td>
<td>Returns a String containing a character representation of the argument.</td>
</tr>
<tr>
<td>intern()</td>
<td></td>
<td>Produces one and only one String reference per unique character sequence.</td>
</tr>
</tbody>
</table>
public class SimpleFormat {
    public static void main(String[] args) {
        int x = 5;
        double y = 5.332542;
        // The old way:
        System.out.println("Row 1: [" + x + " " + y + "]");
        // The new way:
        System.out.format("Row 1: [%d %f]\n", x, y);
        // or
        System.out.printf("Row 1: [%d %f]\n", x, y);
    }
}
The Formatter class

%[argument_index$][flags][width][.precision]conversion

- Need to control the minimum size of a field.
  - This can be accomplished by specifying a \textit{width}.
  - The \textbf{Formatter} guarantees that a field is at least a certain number of characters wide by padding it with spaces.
  - By default
    - the data is right justified
    - this can be overridden by including a ‘-’ in the flags section.
import java.util.*;
public class Receipt {
  private double total = 0;
  private Formatter f = new Formatter(System.out);
  public void printTitle() {
    f.format("%-15s %5s %10s
", "Item", "Qty", "Price");
    f.format("%-15s %5s %10s
", "----", "---", "-----");
  }
  public void print(String name, int qty, double price) {
    f.format("%-15.15s %5d %10.2f
", name, qty, price);
    total += price;
  }
  public void printTotal() {
    f.format("%-15s %5s %10.2f
", "Tax", "", total*0.06);
    f.format("%-15s %5s %10s
", "----", "---", "-----");
    f.format("%-15s %5s %10.2f
", "Total", "", total * 1.06);
  }
}

public static void main(String[] args) {
  Receipt receipt = new Receipt();
  receipt.printTitle();
  receipt.print("Jack’s Magic Beans", 4, 4.25);
  receipt.print("Princess Peas", 3, 5.1);
  receipt.print("Three Bears Porridge", 1, 14.29);
  receipt.printTotal();
}

<table>
<thead>
<tr>
<th>Item</th>
<th>Qty</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack’s Magic Beans</td>
<td>4</td>
<td>4.25</td>
</tr>
<tr>
<td>Princess Peas</td>
<td>3</td>
<td>5.10</td>
</tr>
<tr>
<td>Three Bears Porridge</td>
<td>1</td>
<td>14.29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tax</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.42</td>
</tr>
</tbody>
</table>

| Total               | 25.06|
## Formatter conversions

<table>
<thead>
<tr>
<th>Conversion Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Integral (as decimal)</td>
</tr>
<tr>
<td>c</td>
<td>Unicode character</td>
</tr>
<tr>
<td>b</td>
<td>Boolean value</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
</tr>
<tr>
<td>f</td>
<td>Floating point (as decimal)</td>
</tr>
<tr>
<td>e</td>
<td>Floating point (in scientific notation)</td>
</tr>
<tr>
<td>x</td>
<td>Integral (as hex)</td>
</tr>
<tr>
<td>h</td>
<td>Hash code (as hex)</td>
</tr>
<tr>
<td>%</td>
<td>Literal &quot;%&quot;</td>
</tr>
</tbody>
</table>
String.format()

- **String.format( )**
  - a **static** method which takes all the same arguments as **Format matter’s format( )** but returns a **String**.
  - It can come in handy when you only need to call **format( )** once:

    ```java
    public class DatabaseException extends Exception {
        public DatabaseException(int transactionID, int queryID, String message) {
            super(String.format("(t%d, q%d) %s", transactionID, queryID, message));
        }
        public static void main(String[] args) {
            try {
                throw new DatabaseException(3, 7, "Write failed");
            } catch(Exception e) {
                System.out.println(e);
            }
        }
    }
    ```

    ```
    >>> DatabaseException: (t3, q 7) Write failed
    ```
# Creating regular expressions

<table>
<thead>
<tr>
<th>Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>The specific character B</td>
</tr>
<tr>
<td>\xhh</td>
<td>Character with hex value oxhh</td>
</tr>
<tr>
<td>\uhhhhh</td>
<td>The Unicode character with hex representation oxhhhh</td>
</tr>
<tr>
<td>t</td>
<td>Tab</td>
</tr>
<tr>
<td>n</td>
<td>Newline</td>
</tr>
<tr>
<td>r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>f</td>
<td>Form feed</td>
</tr>
<tr>
<td>e</td>
<td>Escape</td>
</tr>
</tbody>
</table>
Creating regular expressions Cont’d

<table>
<thead>
<tr>
<th>Character Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
</tr>
<tr>
<td>[abc]</td>
</tr>
<tr>
<td>[^abc]</td>
</tr>
<tr>
<td>[a-zA-Z]</td>
</tr>
<tr>
<td>[abc[hij]]</td>
</tr>
<tr>
<td>[a-zA-Z] &amp; [hij]</td>
</tr>
<tr>
<td>\s</td>
</tr>
<tr>
<td>\S</td>
</tr>
<tr>
<td>\d</td>
</tr>
<tr>
<td>\D</td>
</tr>
<tr>
<td>\w</td>
</tr>
<tr>
<td>\W</td>
</tr>
</tbody>
</table>
Creating regular expressions

### Logical Operators

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XY</td>
<td>X followed by Y</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>(X)</td>
<td>A <em>capturing group</em>. You can refer to the <em>i</em>th captured group later in the expression with \i.</td>
</tr>
</tbody>
</table>

### Boundary Matchers

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Beginning of a line</td>
</tr>
<tr>
<td>$</td>
<td>End of a line</td>
</tr>
<tr>
<td>\b</td>
<td>Word boundary</td>
</tr>
<tr>
<td>\B</td>
<td>Non-word boundary</td>
</tr>
<tr>
<td>\G</td>
<td>End of the previous match</td>
</tr>
</tbody>
</table>
Basics of Regular Expressions

• Regular expression backslash
  — \`

• Digit
  — \d

• Literal backslash
  — \\\
  
    • Cf., ‘\’ in other languages

• Number N possibly being preceded by a minus sign
  — -?N

• One or more of the preceding expression
  — +

• Possibly a minus sign, followed by one or more digits
  — -?\d+
More Basics and Integer Match Example

- Grouping
  - ()
- OR
  - |
- Possibly starting with a + or a -
  - (-|\|+)?

```java
public class IntegerMatch {
    public static void main(String[] args) {
        System.out.println("-1234".matches("-? \d+"));
        System.out.println("5678".matches("-? \d+"));
        System.out.println("+911".matches("-? \d+"));
        System.out.println("+911".matches((-|\|+)\d+"));
    }
}
```
import java.util.*;
public class Splitting {
    public static String knights =
        "Then, when you have found the shrubbery, you must " +
        "cut down the mightiest tree in the forest... " +
        "with... a herring!";
    public static void split(String regex) {
        System.out.println(Arrays.toString(knights.split(regex)));
    }
    public static void main(String[] args) {
        split(" "); // Doesn’t have to contain regex chars
        split("\\W+"); // Non-word characters
        split("n\\W+"); // ‘n’ followed by non-word characters
    }
}

[Then, when, you, have, found, the, shrubbery, you, must, cut, down, the, mightiest, tree, in, the, forest..., with..., a, herring!]
[Then, when, you, have, found, the, shrubbery, you, must, cut, down, the, mightiest, tree, in, the, forest, with, a, herring]
[The, whe, you have found the shrubbery, you must cut dow, the mightiest tree i, the forest... with... a herring!]

Then, when you have located the shrubbery, you must cut down the mightiest tree in the forest... with... a herring!
Then, when you have found the banana, you must cut down the mightiest banana in the forest... with... a banana!

public class Rudolph {
public static void main(String[] args) {
    for(String pattern : new String[] { "Rudolph",
                                         "[rR]udolph", "[rR][aeiou][a-z]ol.*", "R.*" })
    System.out.println("Rudolph".matches(pattern));
}
}
Scanner

- The **Scanner** class
  - added in Java SE5
  - relieves much of the burden of scanning input
- The **Scanner** constructor
  - It can take just about any kind of input object, including a **File** object (which will also be covered in the **I/O** chapter), an **InputStream**, a **String**, or in this case a **Readable**
- With **Scanner**, the input, tokenizing, and parsing are all ensconced in various different kinds of "next" methods.
- A plain **next( )**
  - returns the next **String** token
  - there are "next" methods for all the primitive types (except **char**) as well as for **BigDecimal** and **BigInteger**.
- All of the "next" methods **block**
  - they will return only after a complete data token is available for input.
import java.io.*;
public class SimpleRead {
    public static BufferedReader input = new BufferedReader(new StringReader("Sir Robin of Camelot
22 1.61803"));
    public static void main(String[] args) {
        try {
            System.out.println("What is your name?");
            String name = input.readLine();
            System.out.println(name);
            System.out.println("How old are you? What is your favorite double?");
            System.out.println("(input: \<age\> \<double\>)");
            String numbers = input.readLine();
            System.out.println(numbers);
            String[] numArray = numbers.split(" ");
            int age = Integer.parseInt(numArray[0]);
            double favorite = Double.parseDouble(numArray[1]);
            System.out.format("Hi %s.  
", name);
            System.out.format("In 5 years you will be \%d.  
", age + 5);
            System.out.format("My favorite double is \%f.  
", favorite / 2);
        } catch (IOException e) {
            System.out.println("IOException");
        }
    }
}
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22 1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
import java.util.*;
public class BetterRead {
    public static void main(String[] args) {
        Scanner stdin = new Scanner(SimpleRead.input);
        System.out.println("What is your name?");
        String name = stdin.nextLine();
        System.out.println(name);
        System.out.println("How old are you? What is your favorite double? (input: <age> <double>)
        int age = stdin.nextInt();
        double favorite = stdin.nextDouble();
        System.out.println(age);
        System.out.println(favorite);
        System.out.format("Hi %s.\n", name);
        System.out.format("In 5 years you will be %d.\n", age + 5);
        System.out.format("My favorite double is %f.\n", favorite / 2);
    }
}
import java.util.*;

public class ScannerDelimiter {
    public static void main(String[] args) {
        Scanner scanner = new Scanner("12, 42, 78, 99, 42");
        scanner.useDelimiter("\s*,\s*");
        while(scanner.hasNextInt())
            System.out.println(scanner.nextInt());
    }
}

>>
12
42
78
99
42
StringTokenizer

• StringTokenizer
  – Before regular expressions (in J2SE1.4) or the Scanner class (in Java SE5)
  – the way to split a string into parts was to "tokenize"
  – But now it’s much easier and more succinct to do the same thing with regular expressions or the Scanner class.
import java.util.*;

public class ReplacingStringTokenizer {
    public static void main(String[] args) {
        String input = "But I’m not dead yet! I feel happy!";
        StringTokenizer stoke = new StringTokenizer(input);
        while (stoke.hasMoreElements())
            System.out.print(stoke.nextToken() + " ");
        System.out.println();
        System.out.println(Arrays.toString(input.split(" ")));
        Scanner scanner = new Scanner(input);
        while (scanner.hasNext())
            System.out.print(scanner.next() + " ");
    }
}

>>
But I’m not dead yet! I feel happy!
[But, I’m, not, dead, yet!, I, feel, happy!]
But I’m not dead yet! I feel happy!
Arrays

• Most efficient way to hold references to objects
• Limitation: size of an array is fixed
• Benefits
  – Array knows what type it holds, compile-time type checking
  – Knows its size, you can ask
Returning an Array

• Returning Java array == returning a reference
  – Reference knows the type of the array
  – Doesn’t matter where or how array is created
  – Array is around as long as needed, GC cleans up
Arrays of Primitives

• Arrays can hold primitive types directly
• Containers can only hold references
• Can use “wrapper” classes to put primitives into containers, but that’s read only
java.util.Arrays

• Algorithms for array processing:
  – `binarySearch( )`
  – `equals( )`
  – `fill( )`
    • The same object duplicated
  – `sort( )`
    • Unstable Quicksort for primitives
    • Stable merge sort for Objects

• Overloaded for `Object` and all primitives
Summary

• Array associates numerical indices to objects
  – Holds objects of a known type
  – Fixed size