9

Using Strings, String Buffer, Wrapper, and Text-Formatting Classes

Objectives

After completing this lesson, you should be able to do the following:

- Create strings in Java
- Perform operations on strings
- Compare `String` objects
- Use the conversion methods that are provided by the predefined wrapper classes
- Use the formatting classes

Strings in Java

- `String` is a class.
- A `String` object holds a sequence of characters.
- `String` objects are read-only (immutable); their values cannot be changed after creation.
- The `String` class represents all strings in Java.

Creating Strings

- Assign a double-quoted constant to a `String` variable:

```
String category = "Action";
```

- Concatenate other strings:

```
String empName = firstName + " " + lastName;
```

- Use a constructor:

```
String empName = new String("Bob Smith");
```

Concatenating Strings

- Use the + operator to concatenate strings.

```
System.out.println("Name = " + empName);
```

- You can concatenate primitives and strings.

```
int age = getAge();
System.out.println("Age = " + age);
```

- The `String` class has a `concat()` instance method that can be used to concatenate strings.

Performing Operations on Strings

- Find the length of a string:

```
int length();
```

```
String str = "Comedy";
int len = str.length();
```

- Find the character at a specific index:

```
char charAt(int index);
```

```
String str = "Comedy";
char c = str.charAt(1);
```

Return a substring of a string:

```
String substring
       (int beginIndex,
       int endIndex);
```

```
String str = "Comedy";
String sub =
     str.substring(2,4);
```

Performing More Operations on Strings

- Convert to uppercase or lowercase:

```
String toUpperCase();
String toLowerCase();
```

```
String caps =
      str.toUpperCase();
```

- Trim white space:

```
String trim();
```

```
String nospaces =
      str.trim();
```

- Find the index of a substring:

```
int indexOf (String str);
int lastIndexOf
      (String str);
```

```
int index =
      str.indexOf("me");
```

Comparing `String` Objects

- Use `equals()` if you want case to matter:

```
String passwd = connection.getPassword();
if (passwd.equals("fgHPUw"))… // Case is important
```

- Use `equalsIgnoreCase()` if you want to ignore

```
String cat = getCategory();
if (cat.equalsIgnoreCase("Drama"))…
                    // We just want the word to match
```

- Do not use ==.

Producing Strings from Other Objects

- Use the `Object.toString()` method.
- Your class can override `toString()`.

```
public Class Movie {…
      public String toString () {
            return name + " (" + Year + ")";
      }…
```

- `System.out.println()` automatically calls an object's `toString()` method if a reference is passed

```
Movie mov = new Movie(…);
System.out.println(mov);
```

Producing Strings from Primitives

- Use `String.valueOf()`:

```
String seven = String.valueOf(7);
String onePoint0 = String.valueOf(1.0f);
```

- There is a version of `System.out.println()` for

```
int count;
…
System.out.println(count);
```

Producing Primitives from Strings

- Use the primitive wrapper classes.
- There is one wrapper class for each primitive type:
  - `Integer` wraps the `int` type.
  - `Float` wraps the `float` type.
  - `Character` wraps the `char` type.
  - `Boolean` wraps the `boolean` type.
  - And so on…
- Wrapper classes provide methods to convert a `String` to a primitive and to convert a primitive to a `String`.

Wrapper Class Conversion Methods

Example: Use the methods to process data from fields as they are declared.

```
String qtyVal = "17";
String priceVal = "425.00";
int qty = Integer.parseInt(qtyVal);
float price = Float.parseFloat(priceVal);
float itemTotal = qty * price;
```

Changing the Contents of a String

- Use the `StringBuffer/StringBuilder` class for modifiable strings of characters:

```
public String reverseIt(String s) {
  StringBuffer sb = new StringBuffer();
  for (int i = s.length() - 1; i >= 0; i--)
    sb.append(s.charAt(i));
  return sb.toString();
}
```

- Use `StringBuffer/StringBuilder` if you need to keep adding characters to a string.

  Note: `StringBuffer/StringBuilder` has a `reverse()` method.

Formatting Classes

The `java.text` package contains:

- An abstract class called `Format` with the `format()` method shown in the following example:

```
public abstract class Format … {
  public final String format(Object obj){
  //Formats an object and produces a string.
  }
  …
}
```

- Classes that format locale-sensitive information such as dates, numbers, and messages
  - `DateFormat`, `NumberFormat`, and `MessageFormat`

Formatting Dates

- `DateFormat` is an abstract class for date/time formatting subclasses that formats and parses dates or time in a language-independent manner.
- The date/time formatting subclasses, such as `SimpleDateFormat`, allow for formatting (example: date to text) and parsing (example: text to date).
- `DateFormat` helps you to format and parse dates for any locale.
- `DateFormat` provides many class methods for obtaining default date/time formatters based on the default or a given locale and a number of formatting styles.

DecimalFormat Subclass

The `DecimalFormat` subclass:

- Is a concrete subclass of `NumberFormat` for formatting decimal numbers
- Allows for a variety of parameters and for localization to different number formats
- Uses standard number notation in format

```
public DecimalFormat(String pattern);
```

Using `DecimalFormat` for Localization

- You can use `DecimalFormat` to format decimal numbers into locale-specific strings.
- Using `DecimalFormat` allows you to control the display of:
  - Leading and trailing zeroes
  - Prefixes and suffixes
  - Grouping of separators (thousands)
- Formatting classes offer a great deal of flexibility in the formatting of numbers, but they can make your code more complex.

1. What is the output of each code fragment?

a.

```
String s = new String("Friday");
if(s == "Friday")
   System.out.println("Equal A");
if(s.equals("Friday"))
   System.out.println("Equal B");
```

b.

```
int num = 1234567;
System.out.println(String.valueOf(num).charAt(3));
```

2. What is the output of each code fragment?

a.

```
String s1 = "Monday";
String s2 = "Tuesday";
System.out.println(s1.concat(s2).substring(4,8));
```

b.

```
// s3 begins with 2 spaces and ends with 2 spaces
String s3 = "  Monday  ";
System.out.println(s3.indexOf("day"));
System.out.println(s3.trim().indexOf("day"));
```

A Regular Expression

- Is a string that describes a pattern for matching a set of strings
- Is used for matching, replacing, and splitting strings
- Can be used for validation (for example, checking user input for an email address) as well as for parsing input
- Is a powerful tool for string manipulation

Matching Strings

- The `String` class has the following new methods:
  - `matches(regExpr)`
  - `replaceFirst(regExpr, replacement)`
  - `replaceAll(regExpr, replacement)`
  - `split(regExpr)`
- The easiest way to check whether a string matches a regular expression is to call `String.matches` and pass the regular expression to it.

Replacing and Splitting Strings

- `replaceAll(regExpr, replacement)` returns a new string that replaces all matching substrings with the replacement:

```
System.out.println("Java Java Java".replaceAll("v\\w", "wi"));
```

- `replaceFirst(regExpr, replacement)` returns a new string that replaces the first matching substring with the replacement:

```
System.out.println("Java Java Java".replaceFirst("v\\w",
"wi"));
```

- `split(regExpr)` splits a string into substrings delimited by the matches:

```
String[] tokens = "Java1HTML2Perl".split("\\d");
```

Pattern Matching

Java's `java.util.regex` package supports pattern matching via its `Pattern`, `Matcher`, and `PatternSyntaxException` classes.

- `Pattern` objects, also known as patterns, are compiled regexes.
- `Matcher` objects, or matchers, are engines that interpret patterns to locate matches in character sequences.
- `PatternSyntaxException` objects describe illegal regex patterns.

Regular Expression Syntax

- You need to learn a specific syntax to create regular expressions.
- A regular expression consists of literal characters and special symbols; for example, a regex to match an email address format looks like:

```
"^\\s+@\\s+$"
```

- You can find a list of frequently used regular expression syntax below.

Steps Involved in Matching

1. Define a `Pattern` object against which to do the matching. It typically takes a `String` parameter:

```
Pattern email=Pattern.compile("^\\S+@\\S+$");
```

2. Run the `Matcher`:

```
Matcher fit = email.matcher(stringValue);
```

3. Get the result. Use the Boolean `matches` method to find out:

```
if (fit.matches()) {
```

Guided Practice

1. A US Social Security number is xxx-xx-xxxx, where x is a digit. Describe a regex for the Social Security number.
2. Describe a regex for a telephone number that has the structure (xxx) xxx-xxxx, where x is a digit and the first digit cannot be zero.
3. Suppose that customers' last names contain at most twenty-five characters and that the first letter is in uppercase. Describe the regex pattern for a last name.
4. What does each of the following statements return: true or false?

```
System.out.println("abc".matches("a[\\w]c"));
System.out.println("12Java".matches("[\\d]{2}[\\w]{4}"));
System.out.println("12Java".matches("[\\d]*"));
System.out.println("12Java".matches("[\\d]{2}[\\w]{1,15}"));
```

5. What is the output from each of the following statements?

```
System.out.println("Java".replaceAll("[av]", "KH"));
System.out.println("Java".replaceAll("av", "KH"));
System.out.println("Java".replaceFirst("\\w", "KH"));
System.out.println("Java".replaceFirst("\\w*", "KH"));
System.out.println("Java12".replaceAll("\\d", "KH"));
```

6. The following methods split a string into substrings. Describe what is returned in each case.

```
"Java.split("[a]")
"Java.split("[av]")
"Java#HTML#PHP".split("#")
"JavaTOHTMLToPHP".split("T|H")
```

Summary

In this lesson, you should have learned how to:

- Create strings in Java
- Perform operations on strings
- Compare `String` objects
- Use the conversion methods that are provided by the predefined wrapper classes
- Use the formatting classes
- Use regular expressions for matching, replacing, and splitting strings

Practice 1: Overview

This practice covers the following topics:

- Creating a new `Order` class
- Populating and formatting `orderDate`
- Formatting existing `orderDate` values with the `GregorianCalendar` class
- Formatting `orderTotal`