Think Java

CHAPTER 6: LOOPS AND STRINGS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Computers are often used to automate repetitive tasks, such as searching for text in documents. Repeating tasks without making errors is something that computers do well and people do poorly.
- In this chapter, we'll learn how to use while and for loops to add repetition to your code. We'll also take a first look at String methods and solve some interesting problems.

Topics

Part 1: Loops

- While Loop
- Counting Loop
- For Loop
- Nested Loop

Part 2: Strings

- Loop over Strings
- String Pattern Search, Processing

LECTURE 1

The while statement

While Loop for Repetition

- Using a while statement, we can repeat the same code multiple times:

```
int n = 3;
while (n > 0) {
    System.out.println(n);
    n = n - 1;
}
System.out.println("Blastoff!")
```

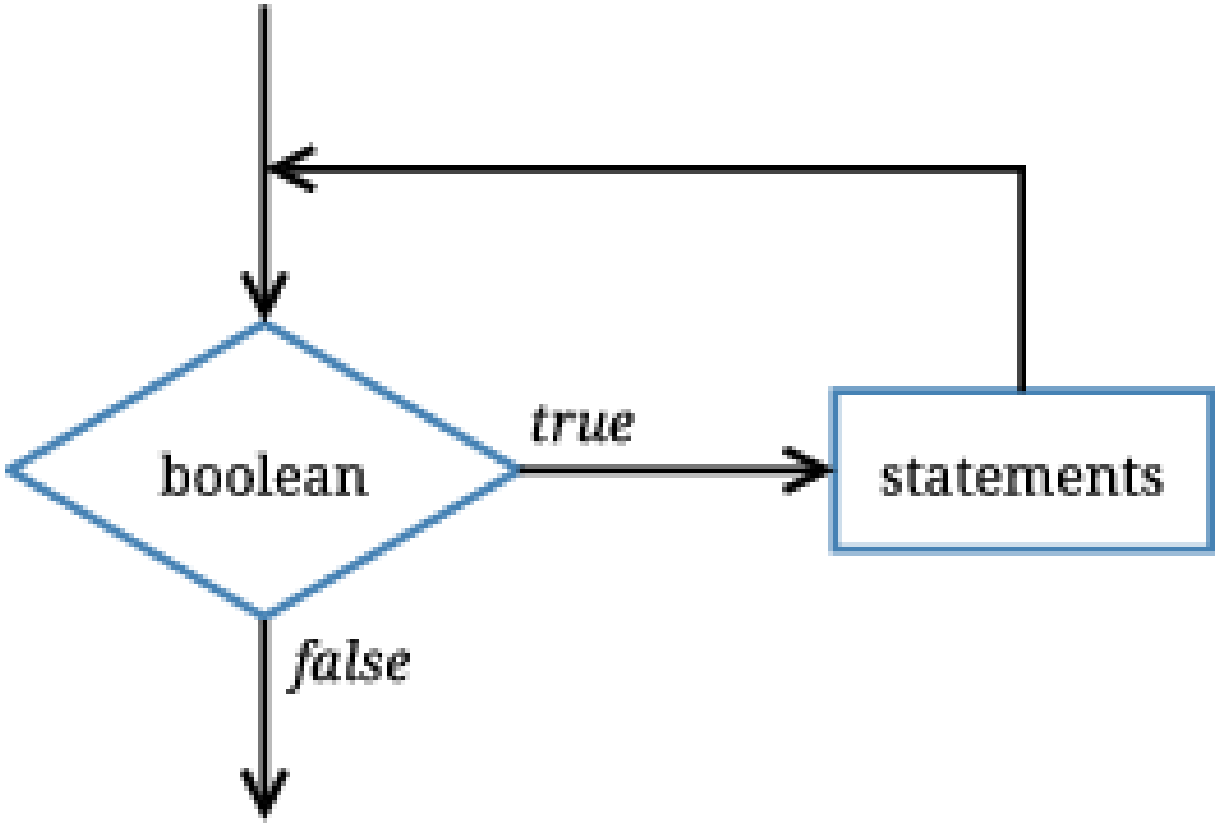
countDown()

- Reading the code in English sounds like: “Start with n set to 3. While n is greater than zero, print the value of n, and reduce the value of n by 1. When you get to zero, print Blastoff!” So the output is:

```
3
2
1
Blastoff!
```

While Loop

- The flow of execution for a while statement is:
  1. Evaluate the condition in parentheses, yielding true or false.
  2. If the condition is false, skip the following statements in braces.
  3. If the condition is true, execute the statements and go back to step 1.
- This type of flow is called a **loop**, because the last step "loops back around" to the first. Figure 6.1 shows this idea using a flowchart.


boolean
true
statements
false

Infinite Loop

- The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**.

```
int n = 3;
while (n > 0) {
    System.out.println(n);
    // n never changes
}
```

- This example will print the number 3 forever, or at least until you terminate the program. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop.

While Loop

- In the first example, we can prove that the loop terminates when n is positive. But in general, it is not so easy to tell whether a loop terminates. For example, this loop continues until n is 1 (which makes the condition false):

```
while (n != 1) {
    System.out.println(n);
    if (n % 2 == 0) {          // n is even
        n = n / 2;
    } else {                   // n is odd
        n = 3 * n + 1;
    }
}
```

While Loop with Loop Variable **n**

- Each time through the loop, the program displays the value of n and then checks whether it is even or odd. If it is even, the value of n is divided by two. If it is odd, the value is replaced by 3n+1. For example, if the starting value is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.
- Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1 and that the program will ever terminate. For some values of n, such as the powers of two, we can prove that it terminates. The previous example ends with such a sequence, starting when n is 16 (or 2^4).

LECTURE 2

Increment and decrement

Indexed Loop and a Loop Counter

- Here is another while loop example; this one displays the numbers 1 to 5.

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;  // add 1 to i
}
```

- Assignments like i = i + 1 don't often appear in loops, because Java provides a more concise way to add and subtract by one. Specifically, **++** is the increment operator; it has the same effect as i = i + 1. And -- is the decrement operator; it has the same effect as i = i - 1.

Jump by 2 Counter

- If you want to increment or decrement a variable by an amount other than 1, you can use += and -=. For example, i += 2 increments i by 2.

```
int i = 2;
while (i <= 8) {
    System.out.print(i + ", ");
    i += 2;  // add 2 to i
}
System.out.println("Who do we appreciate?");
```

- And the output is:

```
2, 4, 6, 8, Who do we appreciate?
```

LECTURE 3

The for statement

For Loop (A Specialized While Loop)

- The loops we have written so far have several elements in common. They start by initializing a variable, they have a condition that depends on that variable, and inside the loop they do something to update that variable.
- Running the same code multiple times is called iteration. This type of loop is so common that there is another statement, the for loop, that expresses it more concisely. For example, we can rewrite the 2-4-6-8 loop this way:

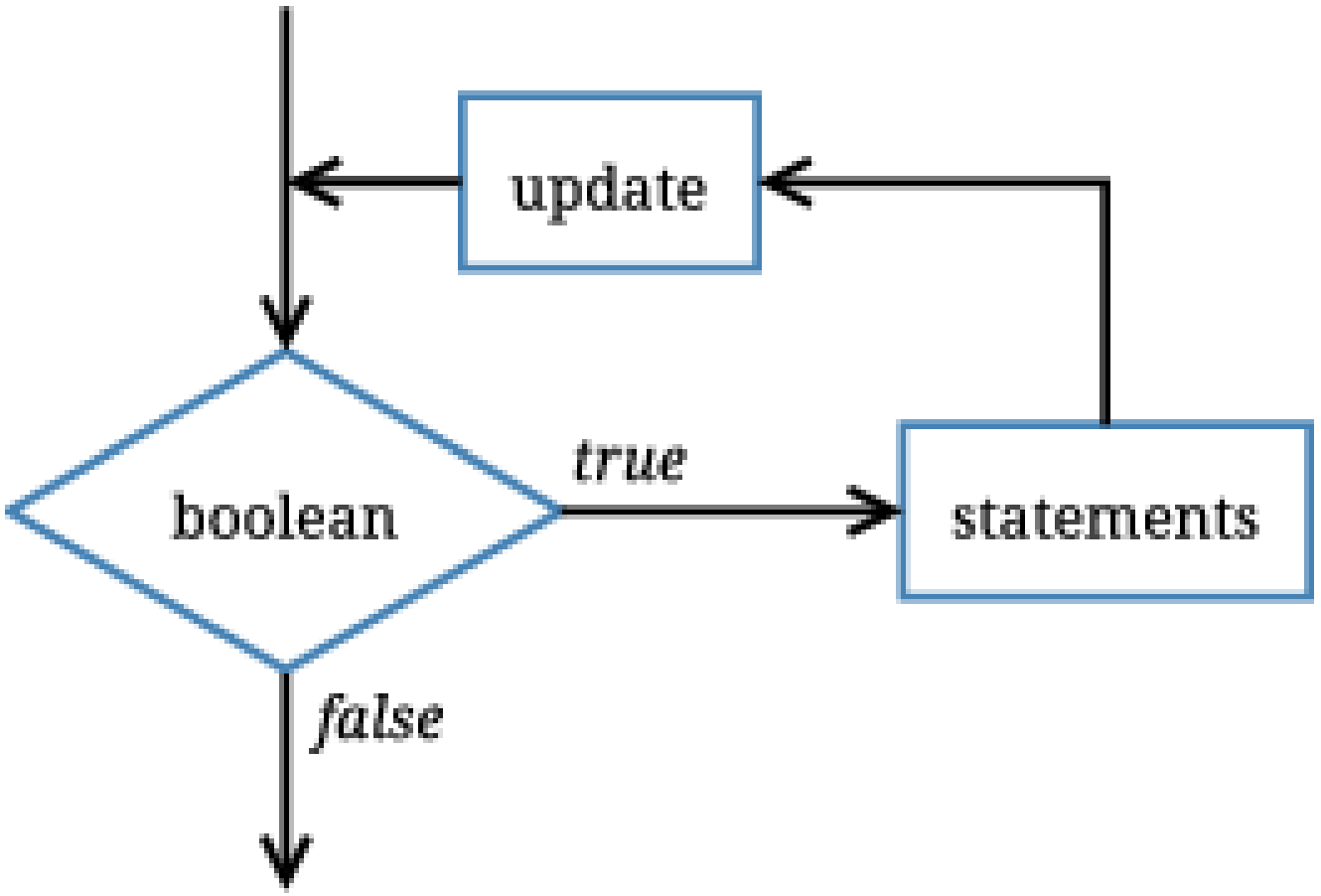
```
for (int i = 2; i <= 8; i += 2) {
    System.out.print(i + ", ");
}
System.out.println("Who do we appreciate?");
```

- for loops have three components in parentheses, separated by semicolons: the initializer, the condition, and the update.

For Loop Structure

1. The *initializer* runs once at the very beginning of the loop. (It is equivalent to the line before the while statement.)
2. The *condition* is checked each time through the loop. If it is false, the loop ends. Otherwise, the body of the loop is executed (again).
3. At the end of each iteration, the *update* runs, and we go back to step 2.

The for loop is often easier to read because it puts all the loop-related statements at the top of the loop. Doing so allows you to focus on the statements in the loop body. Figure 6.2 illustrates for loops with a flowchart


update
boolean
true
statements
false

For Loop

- There is another difference between for loops and while loops: if you declare a variable in the initializer, it only exists inside the for loop. For example:

```
for (int n = 3; n > 0; n--) {
    System.out.println(n);
}
System.out.println("n is now " + n);
// compiler error
```

For Loop

- The last line tries to display n (for no reason other than demonstration) but it won't work. If you need to use a loop variable outside the loop, you have to declare it outside the loop, like this:

```
int n;
for (n = 3; n > 0; n--) {
    System.out.println(n);
}
System.out.println("n is now " + n);
```

- Notice that the for statement does not say int n = 3. Rather, it simply initializes the existing variable n

In-Class Demo Program

- Sum from 1 to 10 while loop version
- Sum from 1 to 10 for loop version

SUM.JAVA

LECTURE 4

Nested loops

Nested Loop

- Like conditional statements, loops can be nested one inside the other. Nested loops allow you to iterate over two variables. For example, we can generate a “multiplication table” like this:

```
for (int x = 1; x <= 10; x++) {
    for (int y = 1; y <= 10; y++) {
        System.out.printf("%4d", x * y);
    }
    System.out.println();
}
```

Name of the Loops (Inner/Outer, i/j)

- Variables like x and y are called loop variables, because they control the execution of a loop. In this example, the first loop (for x) is known as the “outer loop”, and the second loop (for y) is known as the “inner loop”.
- Each loop repeats their corresponding statements 10 times. The outer loop iterates from 1 to 10 only once, but the inner loop iterates from 1 to 10 each of those 10 times. As a result, the printf method is invoked 100 times.

Multiplication Table

- The format specifier \%4d displays the value of x * y padded with spaces so it's four characters wide. Doing so causes the output to align vertically, regardless of how many digits the numbers have:

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

Per Row Operations

- It’s important to realize that the output is displayed row by row. The inner loop displays a single row of output, followed by a newline. The outer loop iterates over the rows themselves.
- Another way to read nested loops, like the ones in this example, is “for each row x, and for each column y, …”

In-Class Demo Program

- Multiplication Table (base 2, 8, 10, 16)

MULTIPLY.JAVA

LECTURE 5

Characters

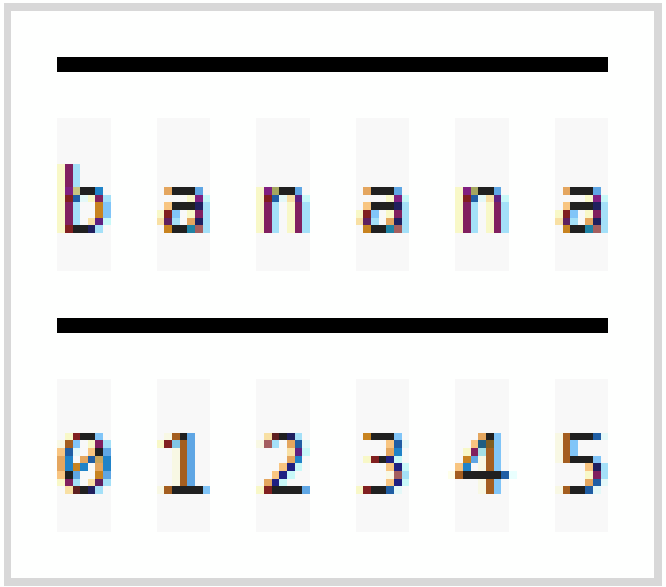
Loops with Characters in a String

- Some of the most interesting problems in computer science involve searching and manipulating text. In the next few sections, we'll discuss how to apply loops to strings. Although the examples are short, the techniques work the same whether you have one word or one million words.
- Strings provide a method named **charAt**. It returns a char, a data type that stores an individual character (as opposed to strings of them).

```
String fruit = "banana";
char letter = fruit.charAt(0);
```

Loops with Characters in a String

- The argument 0 means that we want the character at index 0. String indexes range from 0 to n−1, where n is the length of the string. So the character assigned to letter is b.



Loops with Characters in a String

- Characters work like the other data types we have seen. You can compare them using relational operators:

```
if (letter == 'a') {
    System.out.println('?');
}
```

Loop Over Characters

- Character literals, like 'a', appear in single quotes. Unlike string literals, which appear in double quotes, character literals can only contain a single character. Escape sequences, like '\\t', are legal because they represent a single character.
- The increment and decrement operators also work with characters. So this loop displays the letters of the alphabet:

```
System.out.print("Roman alphabet: ");
for (char c = 'A'; c <= 'Z'; c++) {
    System.out.print(c);
}
System.out.println();
```

Unicode UTF-16

- Java uses Unicode to represent characters, so strings can store text in other alphabets like Cyrillic and Greek, and non-alphabetic languages like Chinese. You can read more about it at http://unicode.org/.

Loop Over a Unicode Region

- In Unicode, each character is represented by a “code point”, which you can think of as an integer. The code points for uppercase Greek letters run from 913 to 937, so we can display the Greek alphabet like this:

```
System.out.print("Greek alphabet: ");
for (int i = 913; i <= 937; i++) {
    System.out.print((char) i);
}
System.out.println();
```

- This example uses a type cast to convert each integer (in the range) to the corresponding character. Try running the code and see what happens

LECTURE 6

String iteration

String Traversal

- The following loop iterates the characters in fruit and displays them, one on each line:

```
for (int i = 0; i < fruit.length(); i++) {
    char letter = fruit.charAt(i);
    System.out.println(letter);
}
```

- Strings provide a method called length that returns the number of characters in the string. Because it is a method, you have to invoke it with the empty argument list, (). When i is equal to the length of the string, the condition becomes false and the loop terminates.

String Traversal

- To find the last letter of a string, you might be tempted to do something like:

```
int length = fruit.length();
char last = fruit.charAt(length);       // wrong!
```

- This code compiles and runs, but invoking the charAt method throws a StringIndexOutOfBoundsException. The problem is that there is no sixth letter in "banana". Since we started counting at 0, the 6 letters are indexed from 0 to 5. To get the last character, you have to subtract 1 from length.

```
int length = fruit.length();
char last = fruit.charAt(length - 1);  // correct
```

String Reversal

- Many string algorithms involve reading one string and building another. For example, to reverse a string, we can add one character at a time:

```
public static String reverse(String s) {
    String r = "";
    for (int i = s.length() - 1; i >= 0; i--) {
        r += s.charAt(i);
    }
    return r;
}
```

- The initial value of r is "", which is the empty string. The loop iterates the letters of s in reverse order. Each time through the loop, it creates a new string and assigns it to r. When the loop exits, r contains the letters from s in reverse order. So the result of reverse("banana") is "ananab"

In-Class Demo Program

- Reverse of String

REVERSE.JAVA

LECTURE 7

The indexOf method

Pattern Search in a String

- To search for a specific character in a string, you could write a for loop and use charAt like in the previous section. However, the String class already provides a method for doing just that.

```
String fruit = "banana";
int index = fruit.indexOf('a'); // returns 1
```

- This example finds the index of 'a' in the string. But the letter appears three times, so it’s not obvious what indexOf should do. According to the documentation, it returns the index of the first appearance.

Pattern Search in a String

- To find subsequent appearances, you can use another version of indexOf, which takes a second argument that indicates where in the string to start looking.

```
int index = fruit.indexOf('a', 2);
// returns 3
```

- To visualize how indexOf and other String methods work, it helps to draw a picture like Figure 6.3. The previous code starts at index 2 (the first 'n') and finds the next 'a', which is at index 3.

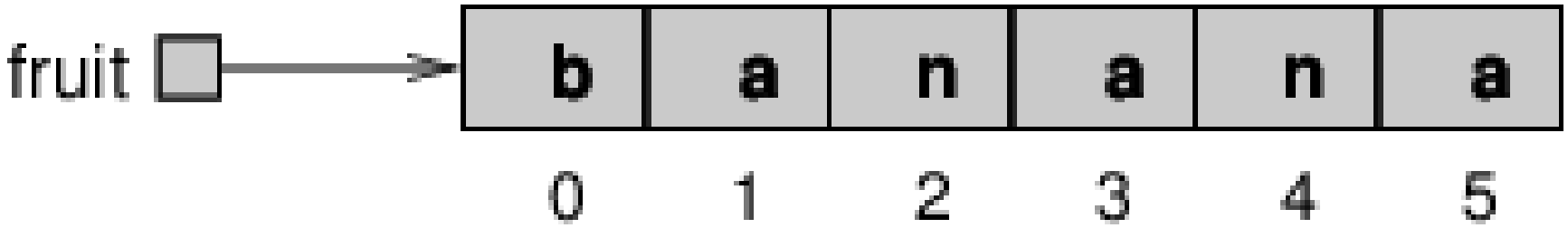


Figure 6.3: Memory diagram for a String of six characters.

Search of a Pattern Location in a String

- If the character happens to appear at the starting index, the starting index is the answer. So fruit.indexOf('a', 5) returns 5. If the character does not appear in the string, indexOf returns -1. Since indexes cannot be negative, this value indicates the character was not found.
- You can also use **indexOf** to search for an entire string, not just a single character. For example, the expression fruit.indexOf("nan") returns 2.

LECTURE 8

String comparison

==, equals(), and compareTo()

- To compare two strings, it may be tempting to use the **==** and **!=** operators.

```
String name1 = "Alan Turing";
String name2 = "Ada Lovelace";
if (name1 == name2) {                    // wrong!
    System.out.println("The names are the same.");
}
```

- This code compiles and runs, and sometimes it gets the answer right. But sometimes it gets the answer wrong. If you give it two different strings that contain the same letters, the condition will be false.

==, equals(), and compareTo()

- The problem is that the == operator checks whether the two variables refer to the same object by comparing the references. We'll learn more about references in the next chapter. The correct way to compare strings is with the equals method, like this:

```
if (name1.equals(name2)) {
System.out.println("The names are the same.");
}
```

- This example invokes equals on name1 and passes name2 as an argument. The equals method returns true if the strings contain the same characters; otherwise it returns false.

==, equals(), and compareTo()

- If the strings differ, we can use compareTo to see which comes first in alphabetical order:

```
int diff = name1.compareTo(name2);
if (diff == 0) {
    System.out.println("The names are the same.");
} else if (diff < 0) {
    System.out.println("name1 comes before name2.");
} else if (diff > 0) {
    System.out.println("name2 comes before name1.");
}
```

==, equals(), and compareTo()

- The return value from compareTo is the difference between the first characters in the strings that are not the same. In the preceding code, compareTo returns positive 8, because the second letter of "Ada" comes before the second letter of "Alan" by 8 letters.

==, equals(), and compareTo()

- If the strings are equal, their difference is zero. If the first string (the one on which the method is invoked) comes first in the alphabet, the difference is negative. Otherwise, the difference is positive.
- Both equals and compareTo are case-sensitive. In Unicode, uppercase letters come before lowercase letters. So "Ada" comes before "ada".

==, equals(), and compareTo()

- The compareTo() method return positive value when name1 has higher alphabetical order than name2. Zero if two are of the same content. Negative if name1 has lower alphabetical value.

```
if (name1.equals(name2)) {
System.out.println("The names are the same.");
}
```

LECTURE 9

Substrings

substring()

- The substring method returns a new string that copies letters from an existing string, starting at the given index.

```
fruit.substring(0) returns "banana"
fruit.substring(2) returns "nana"
fruit.substring(6) returns ""
```

- The first example returns a copy of the entire string. The second example returns all but the first two characters. As the last example shows, substring returns the empty string if the argument is the length of the string.

substring()

- Like most string methods, substring is overloaded. That is, there are other versions of substring that have different parameters. If it's invoked with two arguments, they are treated as a start and end index:

```
fruit.substring(0, 3) returns "ban"
fruit.substring(2, 5) returns "nan"
fruit.substring(6, 6) returns ""
```

- Notice that the character indicated by the end index is not included. Defining substring this way simplifies some common operations. For example, to select a substring with length len, starting at index i, you could write fruit.substring(i, i + len).

LECTURE 10

String formatting

String.format()

- In Section 3.5, we learned how to use System.out.printf to display formatted output. Sometimes programs need to create strings that are formatted a certain way, but not display them immediately, or ever. For example, the following method returns a time string in 12-hour format:

```
public static String timeString(int hour, int minute) {
    String ampm;
    if (hour < 12) {
        ampm = "AM";
        if (hour == 0) {
            hour = 12;  // midnight
        }
    } else {
        ampm = "PM";
        hour = hour - 12;
    }
    return String.format("%02d:%02d %s", hour, minute, ampm);
}
```

String.format()

- String.format takes the same arguments as System.out.printf: a format specifier followed by a sequence of values. The main difference is that System.out.printf displays the result on the screen. String.format creates a new string, but does not display anything.
- In this example, the format specifier \%02d means “two digit integer padded with zeros”, so timeString(19, 5) returns the string "07:05 PM". As an exercise, try writing two nested for loops (in main) that invoke timeString and display all possible times over a 24-hour period.

String.format()

- At some point today, skim through the documentation for String. Knowing what other methods are there will help you avoid reinventing the wheel. The easiest way to find documentation for Java classes is to do a web search for “Java” and the name of the class.

In-Class Demo Program

- Date Format
- European/America

DATEFORMAT.JAVA

Homework

Homework

- Textbook Exercise
- Project 6