Building Java Programs

Chapter 4
Lecture 9: Advanced if/else; Cumulative sum; String/char

reading: 4.2, 4.4 - 4.5
The \textbf{if} statement

\textit{Executes a block of statements only if a test is true}

\begin{verbatim}
if (test) {
    statement;
    ...
    statement;
}
\end{verbatim}

- Example:
  \begin{verbatim}
  double gpa = console.nextDouble();
  if (gpa >= 2.0) {
      System.out.println("Application accepted.");
  }
  \end{verbatim}
The if/else statement

Executes one block if a test is true, another if false

```java
if (test) {
    statement(s);
} else {
    statement(s);
}
```

• Example:
```
double gpa = console.nextDouble();
if (gpa >= 2.0) {
    System.out.println("Welcome to Mars University!");
} else {
    System.out.println("Application denied.");
}
```
Relational expressions

- **if statements** and **for loops** both use logical tests.

```
for (int i = 1; i <= 10; i++)
if (i <= 10)
```

- **These are boolean expressions**, seen in Ch. 5.

- **Tests use relational operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equals</td>
<td>1 + 1 == 2</td>
<td>true</td>
</tr>
<tr>
<td>!=</td>
<td>does not equal</td>
<td>3.2 != 2.5</td>
<td>true</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>10 &lt; 5</td>
<td>false</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>10 &gt; 5</td>
<td>true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>126 &lt;= 100</td>
<td>false</td>
</tr>
<tr>
<td>=&gt;</td>
<td>greater than or equal to</td>
<td>5.0 &gt;= 5.0</td>
<td>true</td>
</tr>
</tbody>
</table>
What's wrong with the following code?

```java
Scanner console = new Scanner(System.in);
System.out.print("What percentage did you earn? ");
int percent = console.nextInt();
if (percent >= 90) {
    System.out.println("You got an A!");
}
if (percent >= 80) {
    System.out.println("You got a B!");
}
if (percent >= 70) {
    System.out.println("You got a C!");
}
if (percent >= 60) {
    System.out.println("You got a D!");
}
if (percent < 60) {
    System.out.println("You got an F!");
}
...
**Nested if/else**

*Chooses between outcomes using many tests*

```java
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else {
    statement(s);
}
```

- **Example:**
  ```java
  if (x > 0) {
      System.out.println("Positive");
  } else if (x < 0) {
      System.out.println("Negative");
  } else {
      System.out.println("Zero");
  }
  ```
**Nested if/else/if**

- If it ends with `else`, exactly one path must be taken.
- If it ends with `if`, the code might not execute any path.

```java
if (test) {
    statement(s);
} else if (test) {
    statement(s);
} else if (test) {
    statement(s);
}
```

- Example:

```java
if (place == 1) {
    System.out.println("Gold medal!");
} else if (place == 2) {
    System.out.println("Silver medal!");
} else if (place == 3) {
    System.out.println("Bronze medal.");
}
```
Nested if structures

- exactly 1 path \((mutually exclusive)\)
  
  ```java
  if (test) {
    statement(s);
  } else if (test) {
    statement(s);
  } else {
    statement(s);
  }
  ```

- 0 or 1 path \((mutually exclusive)\)
  
  ```java
  if (test) {
    statement(s);
  } else if (test) {
    statement(s);
  } else if (test) {
    statement(s);
  }
  ```

- 0, 1, or many paths \((independent tests; not exclusive)\)
  
  ```java
  if (test) {
    statement(s);
  }
  if (test) {
    statement(s);
  }
  if (test) {
    statement(s);
  }
  ```
Which nested \texttt{if/else}?  

\begin{itemize}
  \item (1) \texttt{if/if/if} \quad (2) \texttt{nested if/else} \quad (3) \texttt{nested if/else if}
  \begin{itemize}
    \item Whether a user is lower, middle, or upper-class based on income.
      \begin{itemize}
        \item (2) \texttt{nested if/else if/else}
      \end{itemize}
    \item Whether you made the dean's list (GPA $\geq 3.8$) or honor roll (3.5-3.8).
      \begin{itemize}
        \item (3) \texttt{nested if/else if}
      \end{itemize}
    \item Whether a number is divisible by 2, 3, and/or 5.
      \begin{itemize}
        \item (1) \texttt{sequential if/if/if}
      \end{itemize}
    \item Computing a grade of A, B, C, D, or F based on a percentage.
      \begin{itemize}
        \item (2) \texttt{nested if/else if/else if/else if/else}
      \end{itemize}
  \end{itemize}
\end{itemize}
**Nested if/else question**

Formula for body mass index (BMI):

\[ BMI = \frac{\text{weight}}{\text{height}^2} \times 703 \]

- **Write a program that produces output like the following:**

This program reads data for two people and computes their body mass index (BMI).

Enter next person's information:

height (in inches)? 70.0
weight (in pounds)? 194.25

Enter next person's information:

height (in inches)? 62.5
weight (in pounds)? 130.5

Person 1 BMI = 27.868928571428572 overweight
Person 2 BMI = 23.485824 normal
Difference = 4.3831045714285715

<table>
<thead>
<tr>
<th>BMI</th>
<th>Weight class</th>
</tr>
</thead>
<tbody>
<tr>
<td>below 18.5</td>
<td>underweight</td>
</tr>
<tr>
<td>18.5 - 24.9</td>
<td>normal</td>
</tr>
<tr>
<td>25.0 - 29.9</td>
<td>overweight</td>
</tr>
<tr>
<td>30.0 and up</td>
<td>obese</td>
</tr>
</tbody>
</table>
Advanced if/else

reading: 4.4 - 4.5
Factoring if/else code

- **factoring**: Extracting common/redundant code.
  - Can reduce or eliminate redundancy from if/else code.

**Example:**

```java
if (a == 1) {
    System.out.println(a);
    x = 3;
    b = b + x;
} else if (a == 2) {
    System.out.println(a);
    x = 6;
    y = y + 10;
    b = b + x;
} else { // a == 3
    System.out.println(a);
    x = 9;
    b = b + x;
}
```

```java
System.out.println(a);
```

```java
x = 3 * a;
```

```java
if (a == 2) {
    y = y + 10;
}
```

```java
b = b + x;
```
The "dangling if" problem

- What can be improved about the following code?

```java
if (x < 0) {
    System.out.println("x is negative");
} else if (x >= 0) {
    System.out.println("x is non-negative");
}
```

- The second if test is unnecessary and can be removed:

```java
if (x < 0) {
    System.out.println("x is negative");
} else {
    System.out.println("x is non-negative");
}
```

- This is also relevant in methods that use if with return...
if/else with return

// Returns the larger of the two given integers.
public static int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

- Methods can return different values using if/else
  - Whichever path the code enters, it will return that value.
  - Returning a value causes a method to immediately exit.
  - All paths through the code must reach a return statement.
All paths must return

```java
public static int max(int a, int b) {
    if (a > b) {
        return a;
    }
    // Error: not all paths return a value
}
```

- The following also does not compile:
  ```java
  public static int max(int a, int b) {
      if (a > b) {
          return a;
      } else if (b >= a) {
          return b;
      }
  }
  ```

- The compiler thinks `if/else/if` code might skip all paths, even though mathematically it must choose one or the other.
Logical operators

• Tests can be combined using *logical operators*:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>(2 == 3) &amp;&amp; (-1 &lt; 5)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>!(2 == 3)</td>
<td>true</td>
</tr>
</tbody>
</table>

• "Truth tables" for each, used with logical values *p* and *q*:

| *p* | *q* | *p* && *q* | *p* || *q* |
|-----|-----|------------|--------|
| true| true| true       | true   |
| true| false| false     | true   |
| false| true| false    | true   |
| false| false| false    | false  |

<table>
<thead>
<tr>
<th><em>p</em></th>
<th>!<em>p</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
Evaluating logical expressions

- Relational operators have lower precedence than math; logical operators have lower precedence than relational operators

\[
5 \times 7 \geq 3 + 5 \times (7 - 1) \land 7 \leq 11 \\
5 \times 7 \geq 3 + 5 \times 6 \land 7 \leq 11 \\
35 \geq 3 + 30 \land 7 \leq 11 \\
35 \geq 33 \land 7 \leq 11 \\
true \land true \\
true
\]

- Relational operators cannot be "chained" as in algebra

\[
2 \leq x \leq 10 \\
true \leq 10 \\
\text{(assume that } x \text{ is 15)} \\
\text{Error!}
\]

- Instead, combine multiple tests with \&\& or | |

\[
2 \leq x \land x \leq 10 \\
true \land false \\
false
\]
Logical questions

What is the result of each of the following expressions?

```java
int x = 42;
int y = 17;
int z = 25;
```

- `y < x && y <= z`
- `x % 2 == y % 2 || x % 2 == z % 2`
- `x <= y + z && x >= y + z`
- `!(x < y && x < z)`
- `(x + y) % 2 == 0 || !(z - y) % 2 == 0`

**Answers:** true, false, true, true, false
Cumulative algorithms

reading: 4.2
Adding many numbers

• How would you find the sum of all integers from 1-1000?

// This may require a lot of typing
int sum = 1 + 2 + 3 + 4 + ... ;
System.out.println("The sum is " + sum);

• What if we want the sum from 1 - 1,000,000? Or the sum up to any maximum?
  • How can we generalize the above code?
### Cumulative sum loop

```java
int sum = 0;
for (int i = 1; i <= 1000; i++) {
    sum = sum + i;
}
System.out.println("The sum is " + sum);
```

- **cumulative sum**: A variable that keeps a sum in progress and is updated repeatedly until summing is finished.
  - The `sum` in the above code is an attempt at a cumulative sum.
  - Cumulative sum variables must be declared *outside* the loops that update them, so that they will still exist after the loop.
Cumulative product

- This cumulative idea can be used with other operators:

```java
int product = 1;
for (int i = 1; i <= 20; i++) {
    product = product * 2;
}
System.out.println("2 ^ 20 = " + product);
```

- How would we make the base and exponent adjustable?
Scanner and cumulative sum

- We can do a cumulative sum of user input:

```java
Scanner console = new Scanner(System.in);
int sum = 0;
for (int i = 1; i <= 100; i++) {
    System.out.print("Type a number: ");
    sum = sum + console.nextInt();
}
System.out.println("The sum is "+sum);
```
Cumulative sum question

- Modify the Receipt program from Ch. 2.
  - Prompt for how many people, and each person's dinner cost.
  - Use static methods to structure the solution.

- Example log of execution:

  How many people ate? 4
  Person #1: How much did your dinner cost? 20.00
  Person #2: How much did your dinner cost? 15
  Person #3: How much did your dinner cost? 30.0
  Person #4: How much did your dinner cost? 10.00

  Subtotal: $75.0
  Tax: $6.0
  Tip: $11.25
  Total: $92.25
public class Receipt2 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        double subtotal = meals(console);
        results(subtotal);
    }

    // Prompts for number of people and returns total meal subtotal.
    public static double meals(Scanner console) {
        System.out.print("How many people ate? ");
        int people = console.nextInt();
        double subtotal = 0.0;
        // cumulative sum
        for (int i = 1; i <= people; i++) {
            System.out.print("Person #" + i + ": How much did your dinner cost? ");
            double personCost = console.nextDouble();
            subtotal = subtotal + personCost; // add to sum
        }
        return subtotal;
    }
}

...
Cumulative answer, cont'd.

...  

```java
// Calculates total owed, assuming 8% tax and 15% tip
public static void results(double subtotal) {
    double tax = subtotal * .08;
    double tip = subtotal * .15;
    double total = subtotal + tax + tip;

    System.out.println("Subtotal: "+ subtotal);
    System.out.println("Tax: "+ tax);
    System.out.println("Tip: "+ tip);
    System.out.println("Total: "+ total);
}
```
Write a method `countFactors` that returns the number of factors of an integer.

- `countFactors(24)` returns 8 because 1, 2, 3, 4, 6, 8, 12, and 24 are factors of 24.

Solution:

```java
// Returns how many factors the given number has.
public static int countFactors(int number) {
    int count = 0;
    for (int i = 1; i <= number; i++) {
        if (number % i == 0) {
            count++;
        }
    }
    return count;
}
```
Objects (usage)

- **object**: An entity that contains data and behavior.
  - **data**: variables inside the object
  - **behavior**: methods inside the object

- You interact with the methods; the data is hidden in the object.
- A **class** is a type of objects.

- Constructing (creating) an object:
  
  ```
  Type objectName = new Type(parameters);
  ```

- Calling an object's method:
  
  ```
  objectName.methodName(parameters);
  ```
Strings

- **string**: An object storing a sequence of text characters.
  - Unlike most other objects, a `String` is not created with `new`.

```java
String name = "text";
String name = expression;
```

- Examples:

```java
String name = "Marla Singer";
int x = 3;
int y = 5;
String point = "(" + x + ", " + y + ")";
```
Indexes

• Characters of a string are numbered with 0-based indexes:

```
String name = "Ultimate";
```

<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>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>U</td>
<td>l</td>
<td>t</td>
<td>i</td>
<td>m</td>
<td>a</td>
<td>t</td>
<td>e</td>
</tr>
</tbody>
</table>

• First character's index : 0
• Last character's index : 1 less than the string's length
• The individual characters are values of type char (seen later)
String methods

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>indexOf(str)</code></td>
<td>index where the start of the given string appears in this string (-1 if not found)</td>
</tr>
<tr>
<td><code>length()</code></td>
<td>number of characters in this string</td>
</tr>
<tr>
<td><code>substring(index1, index2)</code></td>
<td>the characters in this string from <code>index1</code> (inclusive) to <code>index2</code> (exclusive); if <code>index2</code> is omitted, grabs till end of string</td>
</tr>
<tr>
<td>or <code>substring(index1)</code></td>
<td></td>
</tr>
<tr>
<td><code>toLowerCase()</code></td>
<td>a new string with all lowercase letters</td>
</tr>
<tr>
<td><code>toUpperCase()</code></td>
<td>a new string with all uppercase letters</td>
</tr>
</tbody>
</table>

- These methods are called using the dot notation:

```java
String starz = "Yeezy & Hova";
System.out.println(starz.length());    // 12
```
String method examples

// index 012345678901
String s1 = "Stuart Reges";
String s2 = "Marty Stepp";
System.out.println(s1.length());  // 12
System.out.println(s1.indexOf("e"));  // 8
System.out.println(s1.substring(7, 10));  // "Reg"
String s3 = s2.substring(1, 7);
System.out.println(s3.toLowerCase());  // "arty s"

• Given the following string:

  // index 0123456789012345678901
  String book = "Building Java Programs";

• How would you extract the word "Java"?
Modifying strings

- Methods like `substring` and `toLowerCase` build and return a new string, rather than modifying the current string.

```java
String s = "Aceyalone";
s.toUpperCase();
System.out.println(s);  // Aceyalone
```

- To modify a variable's value, you must reassign it:

```java
String s = "Aceyalone"
    s = s.toUpperCase()
System.out.println(s);  // ACEYALONE
```
Strings as user input

- Scanner's `next` method reads a word of input as a String.

```java
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
name = name.toUpperCase();
System.out.println(name + " has " + name.length() + " letters and starts with " + name.substring(0, 1));
```

Output:
What is your name? Nas
NAS has 3 letters and starts with N

- The `nextLine` method reads a line of input as a String.

```java
System.out.print("What is your address? ");
String address = console.nextLine();
```
Name border

- Prompt the user for full name
- Draw out the pattern to the left
- This should be resizable. Size 1 is shown and size 2 would have the first name twice followed by last name twice
Strings question

• Write a program that reads two people's first names and suggests a name for their child

Example Output:
Parent 1 first name? Danielle
Parent 2 first name? John
Child Gender? f
Suggested baby name: JODANI

Parent 1 first name? Danielle
Parent 2 first name? John
Child Gender? Male
Suggested baby name: DANIJO
The *equals* method

- Objects are compared using a method named `equals`.

```java
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Lance")) {
    System.out.println("Pain is temporary.");
    System.out.println("Quitting lasts forever.");
}
```

- Technically this is a method that returns a value of type `boolean`, the type used in logical tests.
### String test methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals(str)</td>
<td>whether two strings contain the same characters</td>
</tr>
<tr>
<td>equalsIgnoreCase(str)</td>
<td>whether two strings contain the same characters, ignoring upper vs. lower case</td>
</tr>
<tr>
<td>startsWith(str)</td>
<td>whether one contains other's characters at start</td>
</tr>
<tr>
<td>endsWith(str)</td>
<td>whether one contains other's characters at end</td>
</tr>
<tr>
<td>contains(str)</td>
<td>whether the given string is found within this one</td>
</tr>
</tbody>
</table>

String name = console.next();
if(name.endsWith("Kweli")) {
    System.out.println("Pay attention, you gotta listen to hear.");
} else if(name.equalsIgnoreCase("NaS")) {
    System.out.println("I never sleep 'cause sleep is the cousin of death.");
}

Copyright 2010 by Pearson Education
Type char

- **char**: A primitive type representing single characters.
  - Each character inside a **String** is stored as a **char** value.
  - Literal **char** values are surrounded with apostrophe (single-quote) marks, such as 'a' or '4' or '
' or '\'

- It is legal to have variables, parameters, returns of type **char**
  ```java
cchar letter = 'S';
System.out.println(letter); // S
```

- **char** values can be concatenated with strings.
  ```java
cchar initial = 'P';
System.out.println(initial + " Diddy"); // P Diddy
```
The `charAt` method

- The *chars in a String can be accessed using the `charAt` method.*

```java
String food = "cookie";
char firstLetter = food.charAt(0);  // 'c'
System.out.println(firstLetter + " is for " + food);
System.out.println("That's good enough for me!");
```

- You can use a `for` loop to print or examine each character.

```java
String major = "CSE";
for (int i = 0; i < major.length(); i++) {
    char c = major.charAt(i);
    System.out.println(c);
}
```

**Output:**

C
S
E
char vs. String

- "h" is a String
  'h' is a char (the two behave differently)

- String is an object; it contains methods

  ```java
  String s = "h";
  s = s.toUpperCase();  // 'H'
  int len = s.length();  // 1
  char first = s.charAt(0);  // 'H'
  ```

- char is primitive; you can't call methods on it

  ```java
  char c = 'h';
  c = c.toUpperCase();  // ERROR: "cannot be dereferenced"
  ```

- What is s + 1?  What is c + 1?
- What is s + s?  What is c + c?
char vs. int

- All char values are assigned numbers internally by the computer, called ASCII values.

- Examples:
  'A' is 65, 'B' is 66, ' ' is 32
  'a' is 97, 'b' is 98, '*' is 42

- Mixing char and int causes automatic conversion to int.
  'a' + 10 is 107, 'A' + 'A' is 130

- To convert an int into the equivalent char, type-cast it.
  (char) ('a' + 2) is 'c'
Comparing char values

- You can compare char values with relational operators:
  'a' < 'b' and 'X' == 'X' and 'Q' != 'q'

- An example that prints the alphabet:
  ```java
  for (char c = 'a'; c <= 'z'; c++) {
    System.out.print(c);
  }
  ```

- You can test the value of a string's character:
  ```java
  String word = console.next();
  if (word.charAt(word.length() - 1) == 's') {
    System.out.println(word + " is plural.");
  }
  ```
A Caesar cipher is a simple encryption where a message is encoded by shifting each letter by a given amount.

- e.g. with a shift of 3, $A \rightarrow D$, $H \rightarrow K$, $X \rightarrow A$, and $Z \rightarrow C$

Write a program that reads a message from the user and performs a Caesar cipher on its letters:

Your secret message: Brad thinks Angelina is cute
Your secret key: 3
The encoded message: eudg wklqnv dqjholqd lv fxwh
// This program reads a message and a secret key from the user and
// encrypts the message using a Caesar cipher, shifting each letter.

import java.util.*;

public class SecretMessage {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Your secret message: ");
        String message = console.nextLine();
        message = message.toLowerCase();

        System.out.print("Your secret key: ");
        int key = console.nextInt();

        encode(message, key);
    }

    ...
// This method encodes the given text string using a Caesar cipher, shifting each letter by the given number of places.
public static void encode(String text, int shift) {
    System.out.print("The encoded message: ");
    for (int i = 0; i < text.length(); i++) {
        char letter = text.charAt(i);
        // shift only letters (leave other characters alone)
        if (letter >= 'a' && letter <= 'z') {
            letter = (char) (letter + shift);
            // may need to wrap around
            if (letter > 'z') {
                letter = (char) (letter - 26);
            } else if (letter < 'a') {
                letter = (char) (letter + 26);
            }
        }
        System.out.print(letter);
    }
    System.out.println();
}