Building Java Programs

Chapter 1
Lecture 2: Static Methods

reading: 1.4 - 1.5

(Slides adapted from Stuart Reges, Hélène Martin, and Marty Stepp)
WHAT'S GOING ON? I'M 10 FEET TALL AND BLUE!

AND MY HAIR HAS A LONG BRAID WITH A USB CONNECTOR ON THE END!

I WONDER WHAT WOULD HAPPEN IF I PLUGGED IT IN TO THAT COMPUTER...

IT'S LIKE OUR MINDS ARE ONE! I CAN SPEAK ITS LANGUAGE! THIS IS AWESOME!

OOO - IT WANTS ME TO WRITE SOME OBJECT-ORIENTED CODE...

I HAD THAT "JAVA-TAR" DREAM AGAIN LAST NIGHT.

YOU SAY THAT LIKE I'M SUPPOSED TO BE JEALOUS.
Recall: structure, syntax

Every executable Java program consists of a **class**, that contains a **method** named **main**, that contains the **statements** (commands) to be executed.
Comments

• **comment**: A note written in source code by the programmer to describe or clarify the code.
  • Comments are not executed when your program runs.

• Syntax:
  
  // comment text, on one line
  or,
  /* comment text; may span multiple lines */

• Examples:
  
  // This is a one-line comment.

  /* This is a very long
   * multi-line comment. */
Comments example

/* Suzy Student, CSE 142, Fall 2019
 Displays lyrics*/

public class Lyrics {
    public static void main(String[] args) {
        // first line
        System.out.println("When I first got into magic");
        System.out.println("it was an underground phenomenon");
        System.out.println();

        // second line
        System.out.println("Now everybody's like");
        System.out.println("pick a card, any card");
    }
}
Static methods

reading: 1.4
**Algorithms**

- **algorithm**: A list of steps for solving a problem.

- Example algorithm: "Bake sugar cookies"
  - Mix the dry ingredients.
  - Cream the butter and sugar.
  - Beat in the eggs.
  - Stir in the dry ingredients.
  - Set the oven temperature.
  - Set the timer for 10 minutes.
  - Place the cookies into the oven.
  - Allow the cookies to bake.
  - Spread frosting and sprinkles onto the cookies.
  - ...
Problems with algorithms

- **lack of structure**: Many steps; tough to follow.

- **redundancy**: Consider making a double batch...
  - Mix the dry ingredients.
  - Cream the butter and sugar.
  - Beat in the eggs.
  - Stir in the dry ingredients.
  - Set the oven temperature.
  - Set the timer for 10 minutes.
  - Place the first batch of cookies into the oven.
  - Allow the cookies to bake.
  - Set the timer for 10 minutes.
  - Place the second batch of cookies into the oven.
  - Allow the cookies to bake.
  - Mix ingredients for frosting.
  - ...

Structured algorithms

- **structured algorithm**: Split into coherent tasks.

1. **Make the batter.**
   - Mix the dry ingredients.
   - Cream the butter and sugar.
   - Beat in the eggs.
   - Stir in the dry ingredients.

2. **Bake the cookies.**
   - Set the oven temperature.
   - Set the timer for 10 minutes.
   - Place the cookies into the oven.
   - Allow the cookies to bake.

3. **Decorate the cookies.**
   - Mix the ingredients for the frosting.
   - Spread frosting and sprinkles onto the cookies.
   ...

...
Removing redundancy

- A well-structured algorithm can describe repeated tasks with less redundancy.

1. Make the cookie batter.
   - Mix the dry ingredients.
   - ...

2a. Bake the cookies (first batch).
   - Set the oven temperature.
   - Set the timer for 10 minutes.
   - ...

2b. Bake the cookies (second batch).
   - Repeat Step 2a

3. Decorate the cookies.
   - ...

Static methods

- **static method**: A named group of statements.
  - denotes the *structure* of a program
  - eliminates *redundancy* by code reuse

- **procedural decomposition**: dividing a problem into methods

- Writing a static method is like adding a new command to Java.
Using static methods

1. **Design** (think about) the algorithm.
   - Look at the structure, and which commands are repeated.
   - Decide what are the important overall tasks.

2. **Declare** (write down) the methods.
   - Arrange statements into groups and give each group a name.

3. **Call** (run) the methods.
   - The program's `main` method executes the other methods to perform the overall task.
Declaring a method

*Gives your method a name so it can be executed*

- **Syntax:**

  ```java
  public static void name() {
  statement;
  statement;
  ...
  statement;
  }
  ```

- **Example:**

  ```java
  public static void printWarning() {
  System.out.println("This product causes cancer");
  System.out.println("in lab rats and humans.");
  }
  ```
Calling a method

*Executes the method's code*

- **Syntax:**
  ```
  name();
  ```
  - You can call the same method many times if you like.

- **Example:**
  ```
  printWarning();
  ```

- **Output:**
  ```
  This product causes cancer in lab rats and humans.
  ```
public class FreshPrince {
    public static void main(String[] args) {
        rap();                          // Calling (running) the rap method
        System.out.println();
        rap();                          // Calling the rap method again
        rap();
    }

    // This method prints the lyrics to my favorite song.
    public static void rap() {
        System.out.println("Now this is the story all about how");
        System.out.println("My life got flipped turned upside-down");
    }
}

Output:
Now this is the story all about how
My life got flipped turned upside-down

Now this is the story all about how
My life got flipped turned upside-down
public class MethodsExample {
    public static void main(String[] args) {
        message1();
        message2();
        System.out.println("Done with main.");
    }

    public static void message1() {
        System.out.println("This is message1.");
    }

    public static void message2() {
        System.out.println("This is message2.");
        message1();
        System.out.println("Done with message2.");
    }
}

• Output:
  This is message1.
  This is message2.
  This is message1.
  Done with message2.
  Done with main.
When a method is called, the program's execution...
- "jumps" into that method, executing its statements, then
- "jumps" back to the point where the method was called.

```java
public class MethodsExample {
    public static void main(String[] args) {
        message1();
        message2();
        System.out.println("Done with main.");
    }

    public static void message1() {
        System.out.println("This is message1.");
    }

    public static void message2() {
        System.out.println("This is message2.");
        message1();
        System.out.println("Done with message2.");
    }

    public static void message1() {
        System.out.println("This is message1.");
    }
}
```
When to use methods

- Place statements into a static method if:
  - The statements are related structurally, and/or
  - The statements are repeated.

- You should not create static methods for:
  - An individual `println` statement.
  - Only blank lines. (Put blank `printlns in main.)
  - Unrelated or weakly related statements.
    (Consider splitting them into two smaller methods.)
Drawing complex figures with static methods

reading: 1.5
(Ch. 1 Case Study: DrawFigures)
Static methods question

- Write a program to print these figures using methods.
Development strategy

First version (unstructured):

- Create an empty program and `main` method.
- Copy the expected output into it, surrounding each line with `System.out.println` syntax.
- Run it to verify the output.
public class Figures1 {
    public static void main(String[] args) {
        System.out.println("  ______");
        System.out.println(" /   ");
        System.out.println("/  ");
        System.out.println("/    "/);
        System.out.println("/______/"泓); System.out.println();
        System.out.println("  ______");
        System.out.println(" /   ");
        System.out.println("/STOP/");
        System.out.println("/    "/);
        System.out.println("/______/"泓); System.out.println();
        System.out.println("  ______");
        System.out.println(" /   ");
        System.out.println("/   ");
        System.out.println("/    "/);
        System.out.println("/______/"泓); System.out.println();
    }
}
Development strategy 2

Second version (structured, with redundancy):

- Identify the structure of the output.
- Divide the `main` method into static methods based on this structure.
Output structure

The structure of the output:

- initial "egg" figure
- second "teacup" figure
- third "stop sign" figure
- fourth "hat" figure

This structure can be represented by methods:

- egg
- teaCup
- stopSign
- hat
public class Figures2 {
    public static void main(String[] args) {
        egg();
        teaCup();
        stopSign();
        hat();
    }

    public static void egg() {
        System.out.println("  ______");
        System.out.println("  /
          /  ");
        System.out.println(" /
         / ");
        System.out.println(" \ __\____/ ");
        System.out.println();
    }

    public static void teaCup() {
        System.out.println("  \
          \__\____/ ");
        System.out.println(" +--------+");
        System.out.println();
    }
}
Program version 2, cont'd.

...  

public static void stopSign() {
    System.out.println("  ______");
    System.out.println(" /     \\");
    System.out.println("/     \\");
    System.out.println("| STOP | \\");
    System.out.println("\\     \\");
    System.out.println(" \______/ \\");
    System.out.println();
}

public static void hat() {
    System.out.println("  ______");
    System.out.println(" /     \\");
    System.out.println("/     \\");
    System.out.println("+--------+");
    System.out.println();
}
Development strategy 3

Third version (structured, without redundancy):

- Identify redundancy in the output, and create methods to eliminate as much as possible.

- Add comments to the program.
Output redundancy

The redundancy in the output:

- egg top: reused on stop sign, hat
- egg bottom: reused on teacup, stop sign
- divider line: used on teacup, hat

This redundancy can be fixed by methods:

- eggTop
- eggBottom
- line
Program version 3

// Suzy Student, CSE 138, Spring 2094
// Prints several figures, with methods for structure and redundancy.
public class Figures3 {
    public static void main(String[] args) {
        egg();
        teaCup();
        stopSign();
        hat();
    }

    // Draws the top half of an egg figure.
    public static void eggTop() {
        System.out.println("  ______");
        System.out.println(" / \\
        / \\
        / \\
    }

    // Draws the bottom half of an egg figure.
    public static void eggBottom() {
        System.out.println(" \ \\
        \ \\
        \ \\
    }

    // Draws a complete egg figure.
    public static void egg() {
        eggTop();
        eggBottom();
        System.out.println();
    }

    ...
...  
// Draws a teacup figure.
public static void teaCup() {
    eggBottom();
    line();
    System.out.println();
}

// Draws a stop sign figure.
public static void stopSign() {
    eggTop();
    System.out.println("| STOP |");
    eggBottom();
    System.out.println();
}

// Draws a figure that looks sort of like a hat.
public static void hat() {
    eggTop();
    line();
}

// Draws a line of dashes.
public static void line() {
    System.out.println("+-" + "-");
}
Building Java Programs

Chapter 2
Lecture 2: Expressions

reading: 2.1
...Y'know, I could teach you to process your own data!!
Data and expressions

reading: 2.1
Data types

• Internally, computers store everything as 1s and 0s
  104 \rightarrow 01101000
  "hi" \rightarrow 0110100001101001
  h \rightarrow 01101000

• How are h and 104 differentiated?

• type: A category or set of data values.
  • Constrains the operations that can be performed on data
  • Many languages ask the programmer to specify types
  • Examples: integer, real number, string
Java's primitive types

- **primitive types**: 8 simple types for numbers, text, etc.
  - Java also has **object types**, which we'll talk about later

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integers</td>
<td>42, -3, 0, 926394</td>
</tr>
<tr>
<td>double</td>
<td>real numbers</td>
<td>3.1, -0.25, 9.4e3</td>
</tr>
</tbody>
</table>
| char      | single text characters    | 'a', 'X', '?', '
' |
| boolean   | logical values            | true, false         |

- Why does Java distinguish integers vs. real numbers?
Integer or real number?

- Which category is more appropriate?

<table>
<thead>
<tr>
<th>integer ,(int,)</th>
<th>real number ,(double,)</th>
</tr>
</thead>
</table>

1. Temperature in degrees Celsius
2. The population of lemmings
3. Your grade point average
4. A person's age in years
5. A person's weight in pounds
6. A person's height in meters
7. Number of miles traveled
8. Number of dry days in the past month
9. Your locker number
10. Number of seconds left in a game
11. The sum of a group of integers
12. The average of a group of integers

- credit: Kate Deibel for these examples
- [http://www.youtube.com/watch?v=3TdZHffwOF8&t=1m25s](http://www.youtube.com/watch?v=3TdZHffwOF8&t=1m25s) (for #10)
Expressions

- **expression**: A value or operation that computes a value.
  
  - Examples: 
    
    \[
    1 + 4 \times 5 \\
    (7 + 2) \times 6 / 3 \\
    42
    \]
  
    - The simplest expression is a *literal value*.
    - A complex expression can use operators and parentheses.
Arithmetic operators

- **operator**: Combines multiple values or expressions.
  - +  addition
  - -  subtraction (or negation)
  - *  multiplication
  - /  division
  - %  modulus (a.k.a. remainder)

As a program runs, its expressions are *evaluated*.

- 1 + 1 evaluates to 2
- `System.out.println(3 * 4);` prints 12
  - How would we print the text 3 * 4?
Integer division with /

- When we divide integers, the quotient is also an integer.
  - \( 14 \div 4 \) is 3, not 3.5

\[
\begin{array}{cccc}
3 & & & \\
4 & ) & 14 & \\
12 & & & \\
2 & & & \\
\end{array} \quad \begin{array}{cccc}
4 & & & \\
10 & ) & 45 & \\
40 & & & \\
5 & & & \\
\end{array} \quad \begin{array}{cccc}
52 & & & \\
27 & ) & 1425 & \\
135 & & & \\
54 & & & \\
21 & & & \\
\end{array}
\]

- More examples:
  - \( 32 \div 5 \) is 6
  - \( 84 \div 10 \) is 8
  - \( 156 \div 100 \) is 1

- Dividing by 0 causes an error when your program runs.
Integer remainder with \%

- The \% operator computes the remainder from integer division.
  - \(14 \% 4\) is 2
  - \(218 \% 5\) is 3

\[
\begin{array}{c}
4 \) 14 \\
12 \\
2
\end{array}
\]

\[
\begin{array}{c}
5 \) 218 \\
20 \\
18 \\
15 \\
3
\end{array}
\]

- Applications of \% operator:
  - Obtain last digit of a number: \(230857 \% 10\) is 7
  - Obtain last 4 digits: \(658236489 \% 10000\) is 6489
  - See whether a number is odd: \(7 \% 2\) is 1, \(42 \% 2\) is 0
Precedence

- **precedence**: Order in which operators are evaluated.
  - Generally operators evaluate left-to-right.
    
    \[1 - 2 - 3 \text{ is } (1 - 2) - 3 \text{ which is } -4\]

- But \(* / \%\) have a higher level of precedence than \(+ -\)
  
    \[1 + 3 \times 4 \quad \text{is } 13\]
    \[6 + 8 / 2 \times 3 \quad \text{is } 18\]

- Parentheses can force a certain order of evaluation:
  
    \[(1 + 3) \times 4 \quad \text{is } 16\]

- Spacing does not affect order of evaluation
  
    \[1 + 3 \times 4 - 2 \quad \text{is } 11\]
Precedence examples

1 * 2 + 3 * 5 % 4

2 + 3 * 5 % 4

2 + 15 % 4

2 + 3

5

1 + 8 % 3 * 2 - 9

1 + 2 * 2 - 9

1 + 4 - 9

5 - 9

-4
Precedence questions

- What values result from the following expressions?
  - $9 \div 5$
  - $695 \% 20$
  - $7 + 6 \times 5$
  - $7 \times 6 + 5$
  - $248 \% 100 \div 5$
  - $6 \times 3 - 9 \div 4$
  - $(5 - 7) \times 4$
  - $6 + (18 \% (17 - 12))$
Real numbers (type double)

- Examples: 6.022, -42.0, 2.143e17
  - Placing .0 or . after an integer makes it a double.

- The operators + − * / % () all still work with double.
  - / produces an exact answer: 15.0 / 2.0 is 7.5
  - Precedence is the same: () before * / % before + −
Real number example

\[ 2.0 \times 2.4 + 2.25 \times 4.0 \div 2.0 \]

\[ 4.8 \]

\[ + \quad 2.25 \times 4.0 \div 2.0 \]

\[ 4.8 \quad + \quad 9.0 \quad \div \quad 2.0 \]

\[ 4.8 \quad + \quad 4.5 \]

\[ 9.3 \]
Mixing types

- When int and double are mixed, the result is a double.
  - 4.2 * 3 is 12.6

- The conversion is per-operator, affecting only its operands.
  - 7 / 3 * 1.2 + 3 / 2
    - 2 * 1.2 + 3 / 2
      - 2.4 + 3 / 2
        - 2.4 + 1
          - 3.4

  - 2.0 + 10 / 3 * 2.5 - 6 / 4
    - 2.0 + 3 * 2.5 - 6 / 4
      - 2.0 + 7.5 - 6 / 4
        - 2.0 + 7.5 - 1
          - 9.5 - 1
            - 8.5

- 3 / 2 is 1 above, not 1.5.
String concatenation

- **string concatenation**: Using + between a string and another value to make a longer string.

  - "hello" + 42 is "hello42"
  - 1 + "abc" + 2 is "1abc2"
  - "abc" + 1 + 2 is "abc12"
  - 1 + 2 + "abc" is "3abc"
  - "abc" + 9 * 3 is "abc27"
  - "1" + 1 is "11"
  - 4 - 1 + "abc" is "3abc"

- Use + to print a string and an expression's value together.
  
  - System.out.println("Grade: " + (95.1 + 71.9) / 2);
  
  - Output: Grade: 83.5