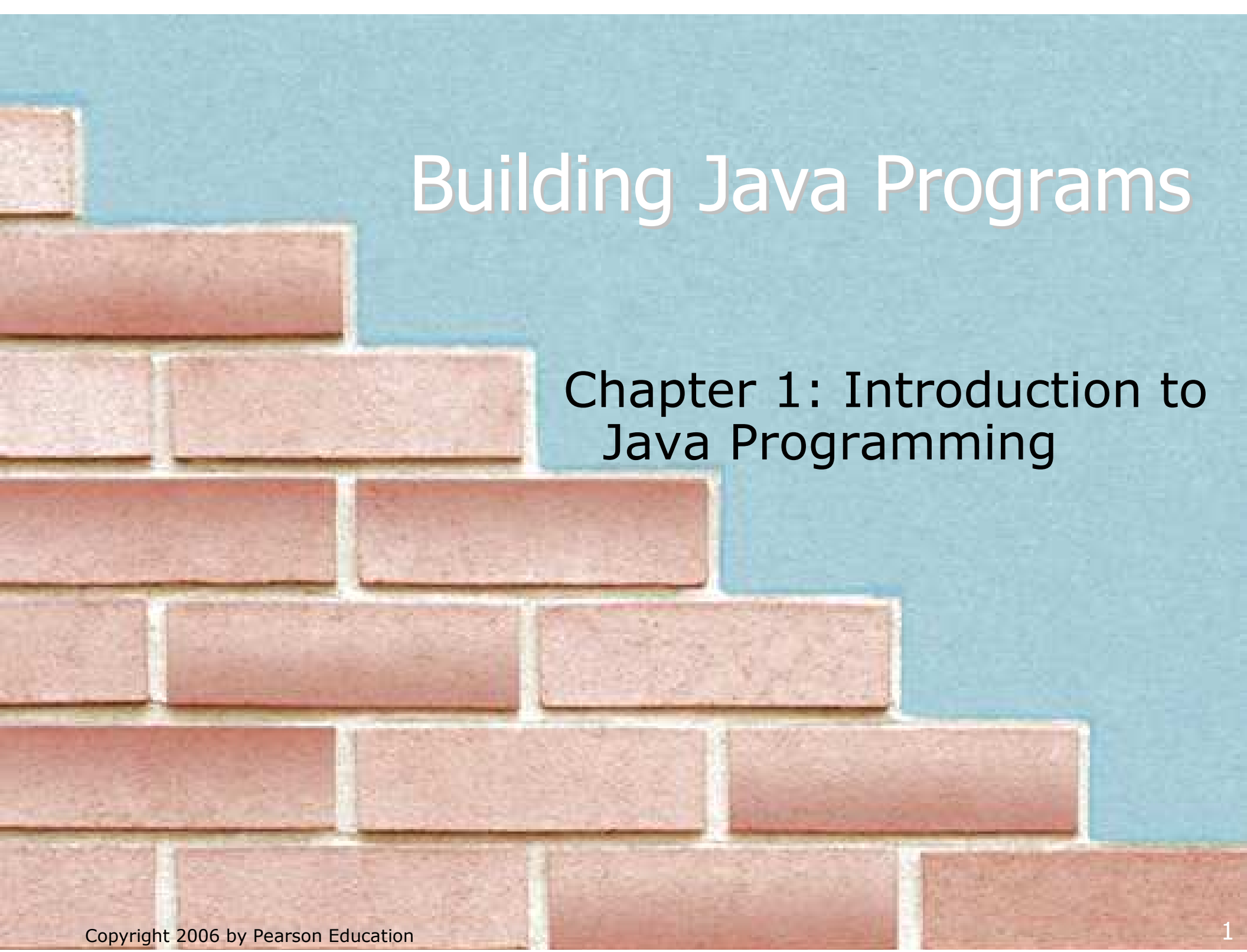
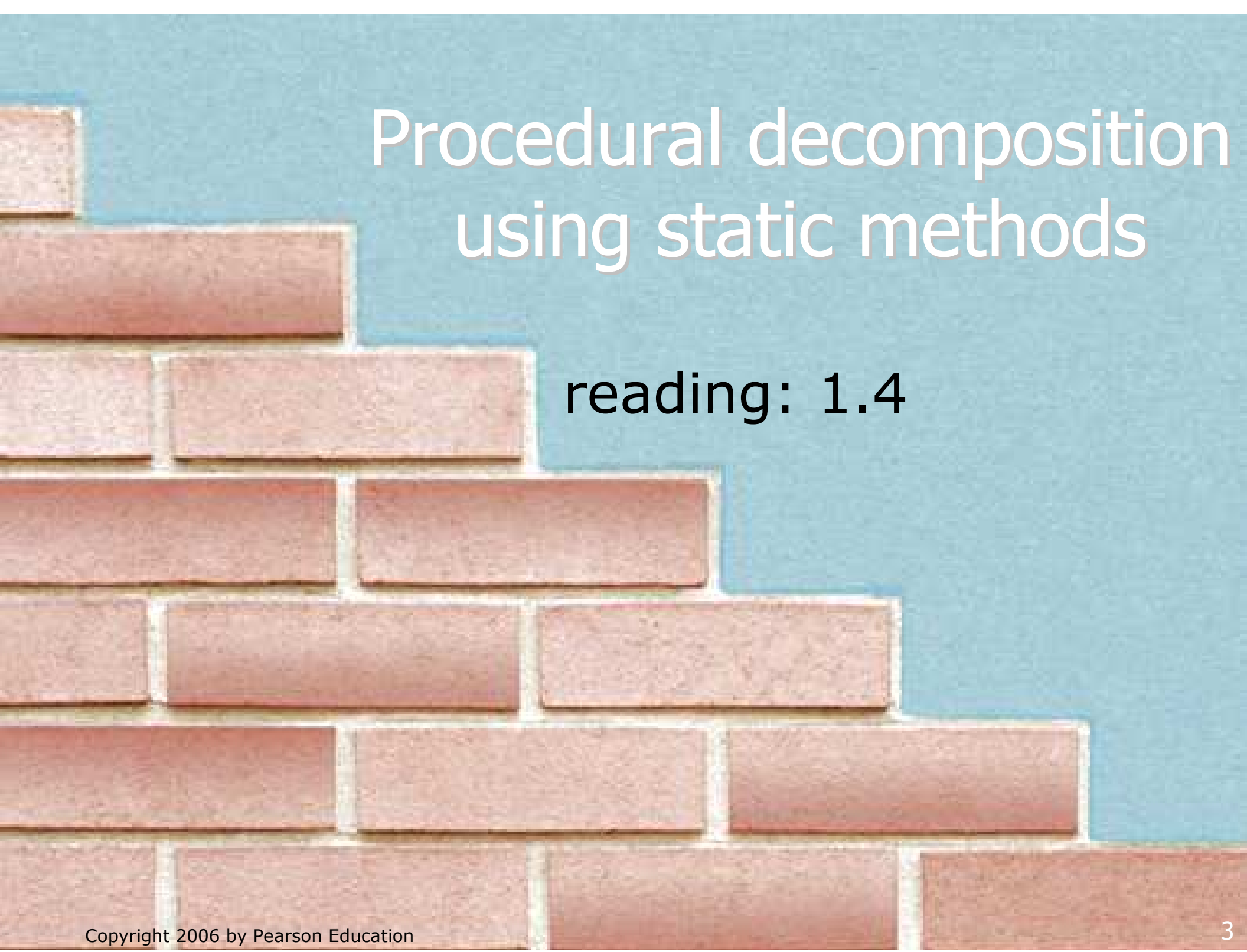


Lecture outline

- procedural decomposition with static methods
  - structured algorithms
  - identifiers, keywords, and comments
  - drawing complex figures

- identifiers, keywords, and comments



Algorithms

- **algorithm**: A list of steps for solving a problem.
- How does one bake sugar cookies?
  (what is the "bake sugar cookies" algorithm?)
  - Mix the dry ingredients.
  - Cream the butter and sugar.
  - Beat in the eggs.
  - Stir in the dry ingredients.
  - Set the oven for the appropriate temperature.
  - Set the timer.
  - Place the cookies into the oven.
  - Allow the cookies to bake.
  - Mix the ingredients for the frosting.
  - Spread frosting and sprinkles onto the cookies.
  - ...

Structured algorithms

- **structured algorithm**: One broken down into cohesive tasks.
- A structured algorithm for baking sugar cookies:

1. Make the cookie 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 for the appropriate temperature.
- Set the timer.
- Place the cookies into the oven.
- Allow the cookies to bake.

3. Add frosting and sprinkles.

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

...

Redundancy in algorithms

- How would we bake a double batch of sugar cookies?

Unstructured:

- Mix the dry ingredients.
- Cream the butter and sugar.
- Beat in the eggs.
- Stir in the dry ingredients.
- *Set the oven ...*
- *Set the timer.*
- *Place the first batch of cookies into the oven.*
- *Allow the cookies to bake.*
- **Set the oven ...**
- **Set the timer.**
- **Place the second batch of cookies into the oven.**
- **Allow the cookies to bake.**
- Mix ingredients for frosting.

Structured:

- 1. Make the cookie batter.
- *2a. Bake the first batch of cookies.*
- **2b. Bake the second batch of cookies.**
- 3. Add frosting and sprinkles.

- *Observations about the structured algorithm*:
  - It is hierarchical, therefore easier to understand.
  - Higher-level operations help eliminate redundancy.

A program with redundancy

- **redundancy**: Occurrence of the same sequence of commands multiple times in a program.

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

  - We print the same messages twice in the program.

Static methods

- **static method**: A group of statements given a name.
  - **procedural decomposition**: breaking a problem into methods

- using a static method requires two steps:
  1. **declare** it (write down the recipe)
     - write a group of statements and give it a name
  2. **call** it (cook using the recipe)
     - tell our program to execute the method

- static methods are useful for:
  - denoting the *structure* of a larger program in smaller pieces
  - eliminating *redundancy* through reuse

Declaring a static method

- Syntax for *declaring* a static method (writing down the recipe):

```
public class <class name> {

    public static void <method name> () {
        <statement>;
        <statement>;
        ...
        <statement>;
    }
}
```

- Example:

```
public static void printWarning() {
    System.out.println("This product is known to cause");
    System.out.println("cancer in lab rats and humans.");
}
```

Calling a static method

- Syntax for *calling* a static method (cooking using the recipe):
  - In another method such as `main`, write:

    <method name> **();**

- Example:

```
printWarning();
```

- You can call the method multiple times.

```
printWarning();
printWarning();
```

Resulting output:

```
This product is known to cause
cancer in lab rats and humans.
This product is known to cause
cancer in lab rats and humans.
```

A program w/ static method

```
public class TwoMessages {
    public static void main(String[] args) {
        displayMessage();
        System.out.println();
        displayMessage();
    }

    public static void displayMessage() {
        System.out.println("Now this is the story all about how");
        System.out.println("My life got flipped turned upside-down");
    }
}
```

Program's 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
```

Methods calling methods

- One static method can call another:

```
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.
```

Control flow of methods

- When a method is called:
  - the execution "jumps" into that method,
  - executes all of its statements, and then
  - "jumps" back to the statement after the method call.

```
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 static methods

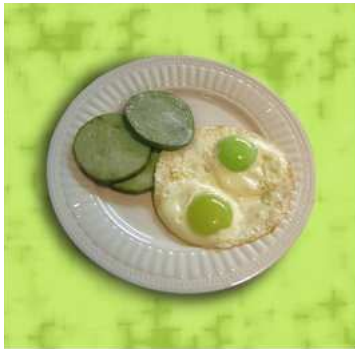
- Place statements into a static method if:
  - The statements are related to each other and form a part of the program's structure, or
  - The statements are repeated in the program.

- You need not create static methods for:
  - Individual statements only occurring once in the program. (A single println in a method does not improve the program.)
  - Unrelated or weakly related statements. (Consider splitting the method into two smaller methods.)
  - Only blank lines. (Blank `println` statements can go in the `main` method.)

Static method questions

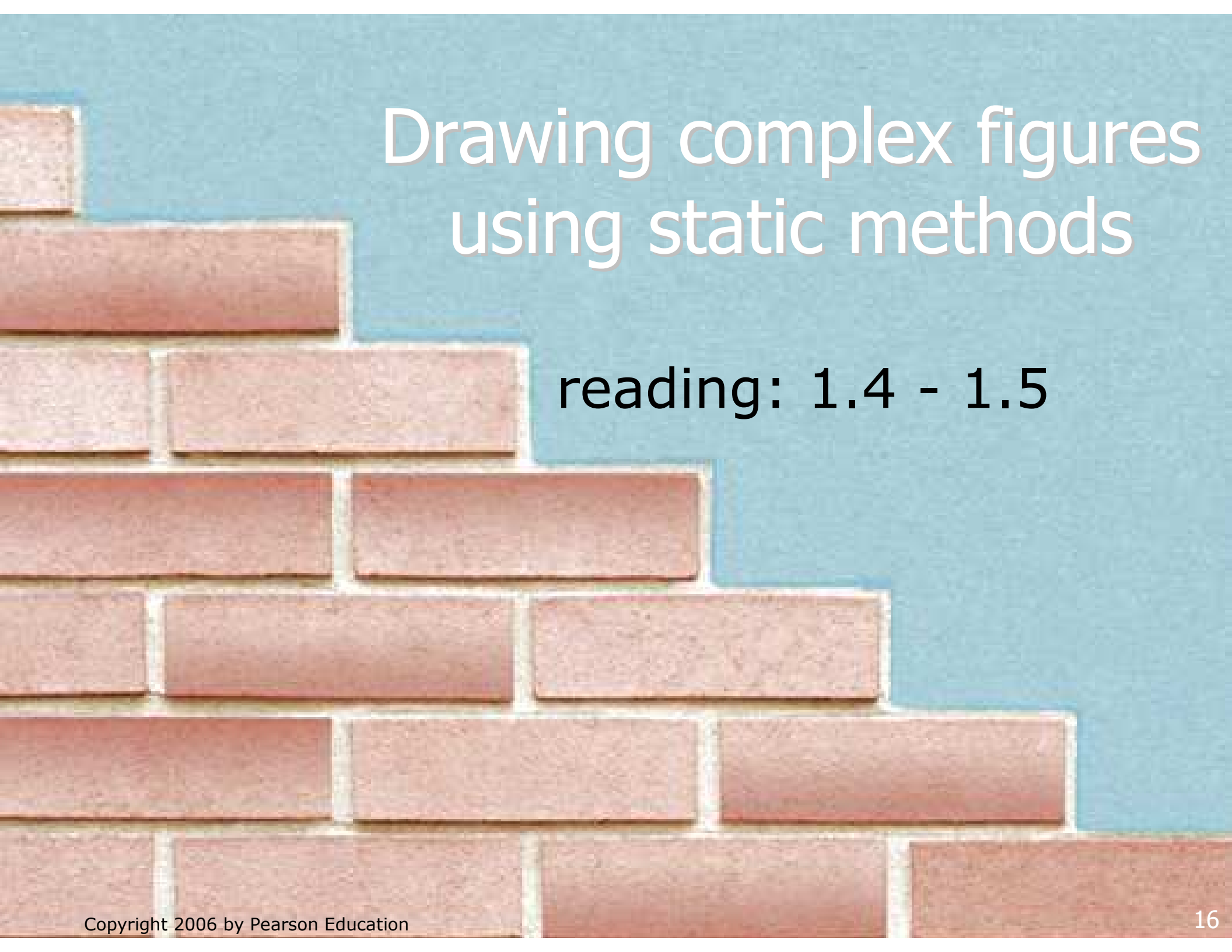
- Write a program that prints the following output to the console. Use static methods as appropriate.

```
I do not like my email spam,
I do not like them, Sam I am!
I do not like them on my screen,
I do not like them to be seen.
I do not like my email spam,
I do not like them, Sam I am!
```



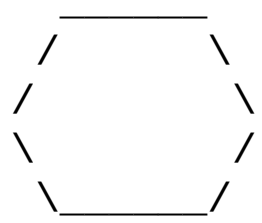
- Write a program that prints the following output to the console. Use static methods as appropriate.

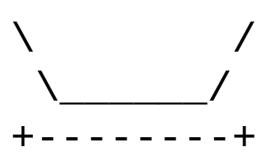
```
Lollipop, lollipop
Oh, lolli lolli lolli

Lollipop, lollipop
Oh, lolli lolli lolli

Call my baby lollipop
```

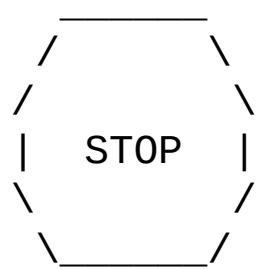


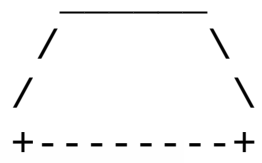
Static methods question

- Write a program to print the following figures. Use static methods for structure and to reduce redundancy.



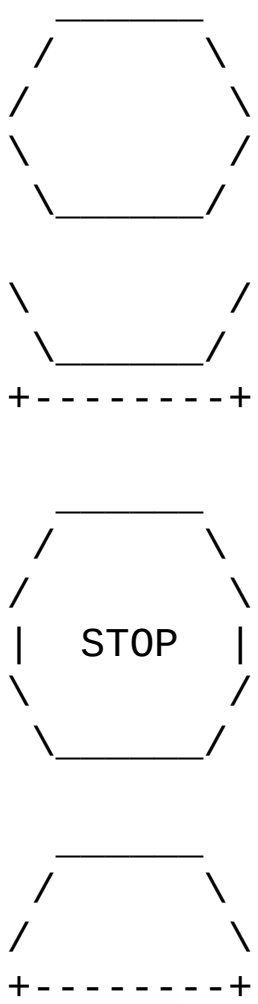






Problem-solving methodology

- Some steps we can use to print complex figures:



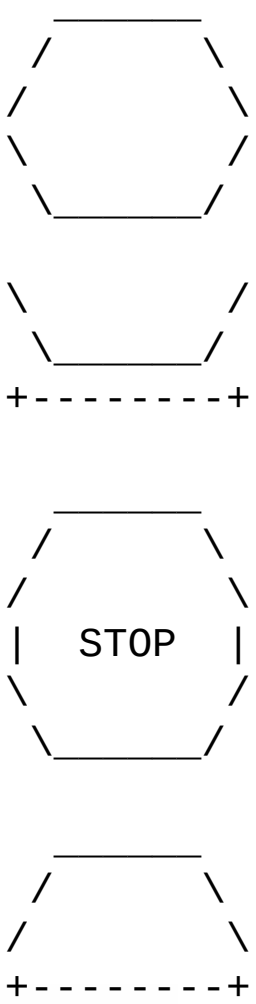
First version of program (unstructured):

- Create an empty program with a skeletal header and `main` method.
- Copy the expected output into it, surrounding each line with `System.out.println` syntax.
- Run our first version and verify that it produces the correct output.

Program, version 1

```
public class Figures1 {
    public static void main(String[] args) {
        System.out.println("  ______");
        System.out.println(" /      \\");
        System.out.println("/        \\");
        System.out.println("\\        /");
        System.out.println(" \\______/");
        System.out.println();
        System.out.println("\\        /");
        System.out.println(" \\______/");
        System.out.println("+--------+");
        System.out.println();
        System.out.println("  ______");
        System.out.println(" /      \\");
        System.out.println("/        \\");
        System.out.println("|  STOP  |");
        System.out.println("\\        /");
        System.out.println(" \\______/");
        System.out.println();
        System.out.println("  ______");
        System.out.println(" /      \\");
        System.out.println("/        \\");
        System.out.println("+--------+");
    }
}
```

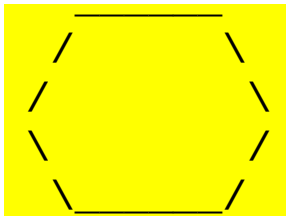
Problem-solving 2

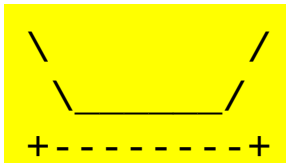


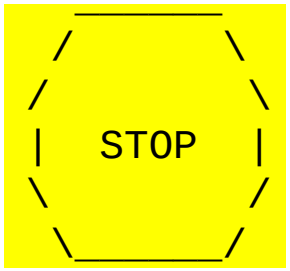
Second version of program (structured with redundancy):

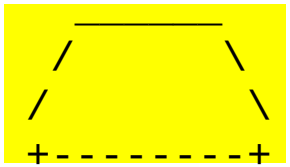
- Identify the structure of the output.
- Divide the `main` method into several static methods based on this structure.

Problem-solving 2 answer









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:

- `drawEgg`
- `drawTeaCup`
- `drawStopSign`
- `drawHat`

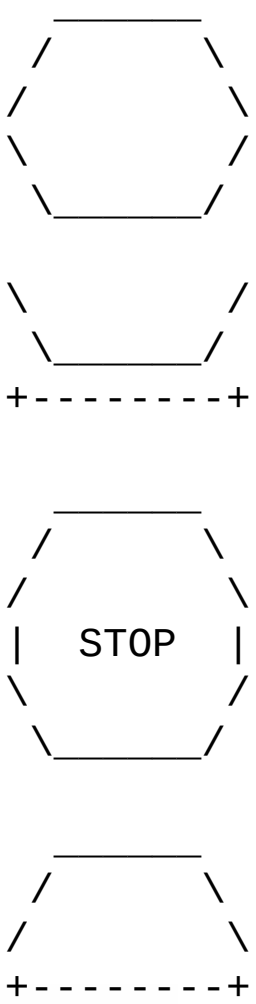
Program, version 2

```
public class Figures2 {
    public static void main(String[] args) {
        drawEgg();
        drawTeaCup();
        drawStopSign();
        drawHat();
    }

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

    public static void drawTeaCup() {
        System.out.println("\\        /");
        System.out.println(" \\______/");
        System.out.println("+--------+");
        System.out.println();
    }
    ...
```

Program, version 2, cont'd.

```
    ...

    public static void drawStopSign() {
        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 drawHat() {
        System.out.println("  ______");
        System.out.println(" /      \\");
        System.out.println("/        \\");
        System.out.println("+--------+");
    }
}
```

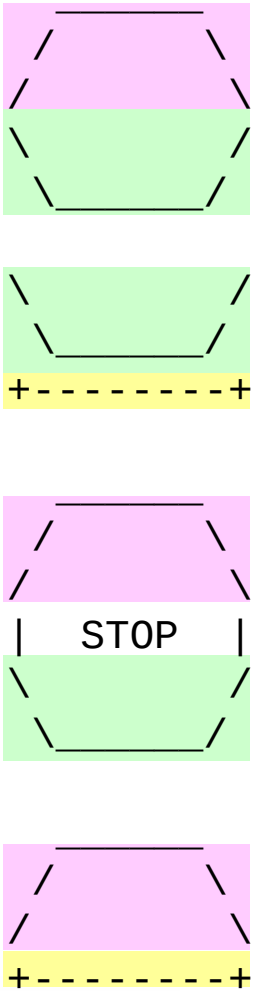
Problem-solving 3



Third version of program (structured without redundancy):

- Identify any redundancy in the output, and further divide the program into static methods to eliminate as much redundancy as possible.
- Add comments to the program to improve its readability.

Problem-solving 3 answer



The redundancy in the output:

- top half of egg: reused on stop sign, hat
- bottom half of egg: reused on teacup, stop sign
- divider line: used on teacup, hat
  - a single line, so making it a method is optional

This redundancy can be fixed by methods:

- `drawEggTop`
- `drawEggBottom`
- `drawLine` (optional)

Program, version 3

```
public class Figures3 {
    public static void main(String[] args) {
        drawEgg();
        drawTeaCup();
        drawStopSign();
        drawHat();
    }
    public static void drawEggTop() {
        System.out.println("  ______");
        System.out.println(" /      \\");
        System.out.println("/        \\");
    }
    public static void drawEggBottom() {
        System.out.println("\\        /");
        System.out.println(" \\______/");
    }
    ...
```

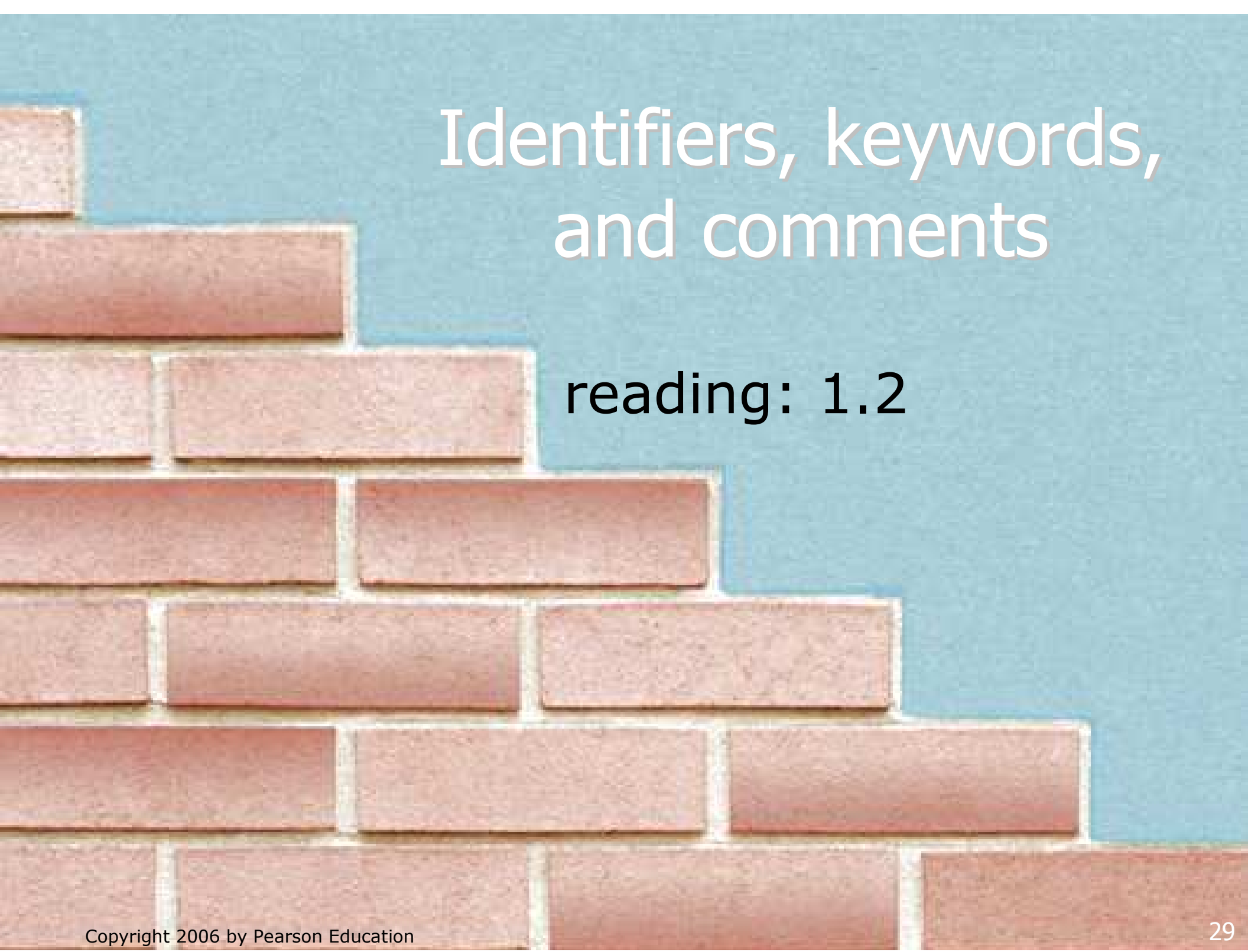
Program, version 3, cont'd.

```
    ...

    public static void drawEgg() {
        drawEggTop();
        drawEggBottom();
        System.out.println();
    }
    public static void drawTeaCup() {
        drawEggBottom();
        System.out.println("+--------+");
        System.out.println();
    }
    public static void drawStopSign() {
        drawEggTop();
        System.out.println("|  STOP  |");
        drawEggBottom();
        System.out.println();
    }
    public static void drawHat() {
        drawEggTop();
        System.out.println("+--------+");
    }
}
```

Another example

- Write a program to print letters spelling "banana". Use static methods for structure and to reduce redundancy.

```
BBBBB
B    B
BBBBB
B    B
BBBBB

 AAAA
A    A
AAAAAA
A    A

N    N
NNN  N
N  NNN
N    N

 AAAA
A    A
AAAAAA
A    A

N    N
NNN  N
N  NNN
N    N

 AAAA
A    A
AAAAAA
A    A
```



Identifiers

- **identifier**: A name given to a piece of data, method, etc.
  - Identifiers allow us to refer to an item later in the program.
  - Identifiers give names to:
    - classes
    - methods
    - variables, constants (seen in Ch. 2)

- Conventions for naming in Java:
  - *classes*: capitalize each word (`ClassName`)
  - *methods*: capitalize each word after the first (`methodName`)
    (variable names follow the same convention)
  - *constants*: all caps, words separated by _ (`CONSTANT_NAME`)

Details about identifiers

- Java identifiers:
  - first character must a letter or _ or $
  - following characters can be any of those or a number
  - identifiers are case-sensitive (`name` is different from `Name`)

- Example Java identifiers:
  - legal: `susan` `second_place` `_myName`
    `TheCure` `ANSWER_IS_42` `$variable`
  - illegal: `me+u` `49er` `question?`
    `side-swipe` `hi there` `ph.d`
    `jim's` `2%milk` `suzy@yahoo.com`
    - can you explain why each of the above identifiers is not legal?

Keywords

- **keyword**: An identifier that you cannot use because it already has a reserved meaning in the Java language.

- Complete list of Java keywords:

```
abstract    default     if          private     this
boolean     do          implements  protected   throw
break       double      import      public      throws
byte        else        instanceof  return      transient
case        extends     int         short       try
catch       final       interface   static      void
char        finally     long        strictfp    volatile
class       float       native      super       while
const       for         new         switch
continue    goto        package     synchronized
```

- You may not use `char` or `while` for the name of a class or method; Java reserves those to mean other things.
  - You could use `CHAR` or `While`, because Java is case-sensitive. However, this could be confusing and is not recommended.

Comments

- **comment**: A note written in the source code by the programmer to make the code easier to understand.
  - Comments are not executed when your program runs.
  - Most Java editors show your comments with a special color.

- Comment, general syntax:

```
/* <comment text; may span multiple lines> */
```

or,

```
// <comment text, on one line>
```

- Examples:

```
/* A comment goes here. */
/* It can even span
   multiple lines. */
// This is a one-line comment.
```

Using comments

- Where to place comments:
  - at the top of each file (also called a "comment header"), naming the author and explaining what the program does
  - at the start of every method, describing its behavior
  - inside methods, to explain complex pieces of code (more useful later)

- Comments provide important documentation.
  - Later programs will span hundreds of lines with many methods.
  - Comments provide a simple description of what each class, method, etc. is doing.
  - When multiple programmers work together, comments help one programmer understand the other's code.

Comments example

```
/* Suzy Student
   CS 101, Fall 2019
   This program prints lyrics from my favorite song! */
public class MyFavoriteSong {
    /* Runs the overall program to print the song
       on the console. */
    public static void main(String[] args) {
        sing();

        // Separate the two verses with a blank line
        System.out.println();

        sing();
    }

    // Displays the first verse of the theme song.
    public static void sing() {
        System.out.println("Now this is the story all about how");
        System.out.println("My life got flipped turned upside-down");
    }
}
```

How to comment: methods

- Do not describe the syntax/statements in detail.
- Instead, provide a short English description of the observed behavior when the method is run.

  - Example:

```
// This method prints the lyrics to the first verse
// of my favorite TV theme song.
// Blank lines separate the parts of the verse.
public static void verse1() {
    System.out.println("Now this is the story all about how");
    System.out.println("My life got flipped turned upside-down");
    System.out.println();
    System.out.println("And I'd like to take a minute,");
    System.out.println("just sit right there");
    System.out.println("I'll tell you how I became the prince");
    System.out.println("of a town called Bel-Air");
}
```

Commented Figures program

```
// Author: Suzy Student
// Prints several figures, with methods for structure and redundancy.
//
public class Figures3 {
    public static void main(String[] args) {
        drawEgg();
        drawTeaCup();
        drawStopSign();
        drawHat();
    }

    // draws redundant part that looks like the top of an egg
    public static void drawEggTop() {
        System.out.println("  ______");
        System.out.println(" /      \\");
        System.out.println("/        \\");
    }

    // draws redundant part that looks like the bottom of an egg
    public static void drawEggBottom() {
        System.out.println("\\        /");
        System.out.println(" \\______/");
    }

    ...
```