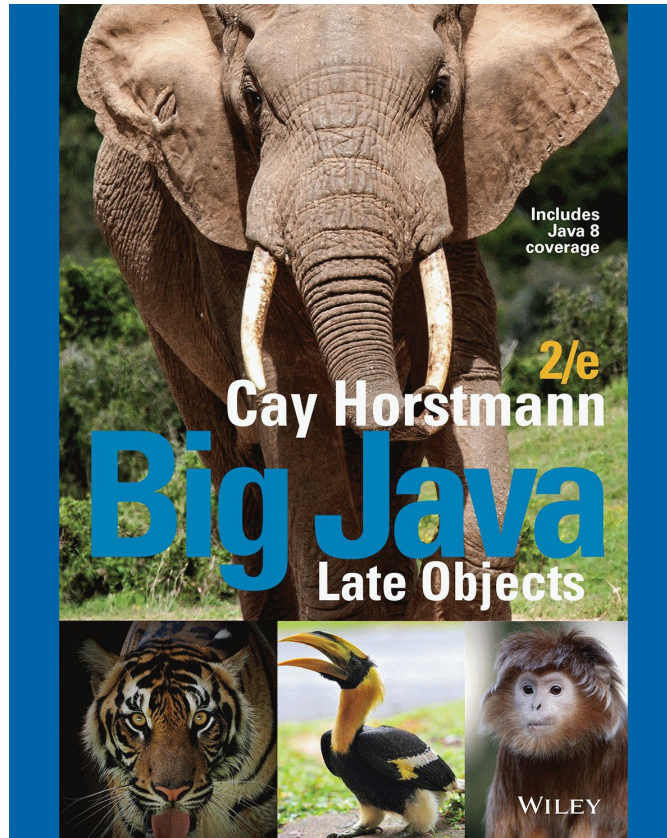


## Chapter 5 - Methods

---



# Chapter Goals

---



- To be able to implement methods
- To become familiar with the concept of parameter passing
- To develop strategies for decomposing complex tasks into simpler ones
- To be able to determine the scope of a variable
- To learn how to think recursively (optional)

# Methods as Black Boxes

---

- A method is a sequence of instructions with a name
  - You declare a method by defining a named block of code

```
public static void square(int x)
{
    int result = x * x;
    return result;
}
```

- You call a method in order to execute its instructions

```
public static void main(String[] args)
{
    double result = Math.pow(2, 3);
    . . .
}
```

# What Is a Method?

---

- Some methods you have already used are:

- `Math.pow()`
- `String.length()`
- `Character.isDigit()`
- `Scanner.nextInt()`
- `main()`

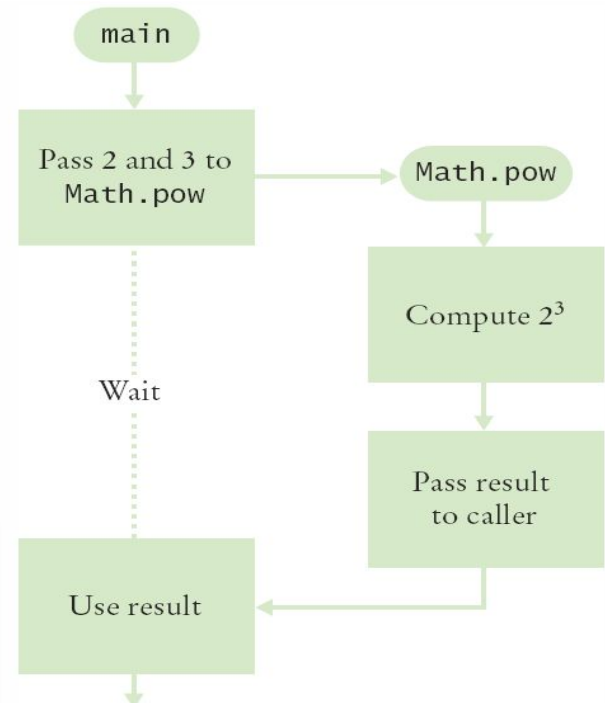
- Methods may have:

- A capitalized name and a dot (.) before them
- A method name
  - Follow the same rules as variable names, camelHump style
- ( ) - a set of parenthesis at the end
  - A place to provide the method input information

# Flowchart of Calling a Method

- One method 'calls' another
  - main calls `Math.pow()`
  - Passes two arguments
    - 2 and 3
  - `Math.pow` starts
    - Uses variables (2, 3)
    - Does its job
    - Returns the answer
- main uses result

```
public static void main(String[] args)
{
    double result = Math.pow(2, 3);
    . . .
}
```

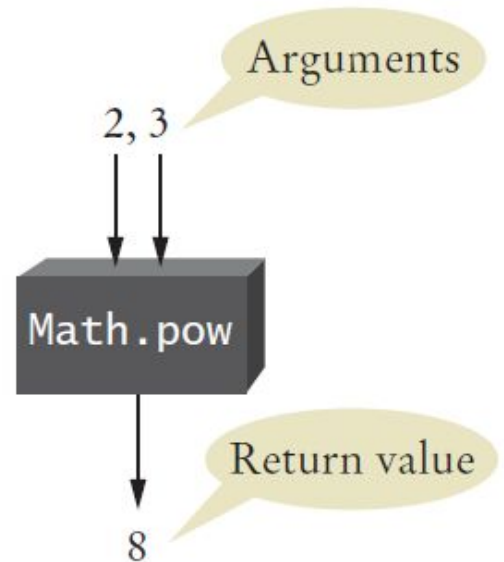


# Arguments and Return Values

---

- `main` 'passes' two arguments (2 and 3) to `Math.pow`
- `Math.pow` calculates and returns a value of 8 to `main`
- `main` stores the return value to variable 'result'

```
public static void main(String[] args)
{
    double result = Math.pow(2,3);
    . . .
}
```



# Black Box Analogy

---

- A thermostat is a 'black box'
  - Set a desired temperature
  - Turns on heater/AC as required
  - You don't have to know how it really works!
    - How does it know the current temp?
    - What signals/commands does it send to the heater or A/C?
- Use methods like 'black boxes'
  - Pass the method what it needs to do its job
  - Receive the answer



## Self Check 5.1

---

Consider the method call `Math.pow(3, 2)`. What are the arguments and return values?

**Answer:** The arguments are 3 and 2. The return value is 9.



## Self Check 5.2

---

What is the return value of the method call `Math.pow(Math.pow(2, 2), 2)` ?

**Answer:** The inner call to `Math.pow` returns  $2^2 = 4$ . Therefore, the outer call returns  $4^2 = 16$ .

## Self Check 5.3

---

The `Math.ceil` method in the Java standard library is described as follows: The method receives a single argument `a` of type `double` and returns the smallest `double` value  $\geq a$  that is an integer. What is the return value of `Math.ceil(2.3)` ?

**Answer:** 3.0

## Self Check 5.4

---

It is possible to determine the answer to Self Check 3 without knowing how the `Math.ceil` method is implemented. Use an engineering term to describe this aspect of the `Math.ceil` method.

**Answer:** Users of the method can treat it as a *black box*.

# Implementing Methods

---

- A method to calculate the volume of a cube
  - What does it need to do its job?
  - What does it answer with?
- When declaring a method, you provide a **name for the method**, a **variable for each argument**, and a **type for the result**
  - Pick a name for the method (`cubeVolume`).
  - Declare a variable for each incoming argument
    - (`double sideLength`) (called parameter variables)
  - Specify the type of the return value (`double`)
  - Add modifiers such as `public static`
    - (see Chapter 8)

```
public static double cubeVolume(double sideLength)
```

# Inside the Box

---

- Then write the body of the method
  - The body is surrounded by curly braces { }
  - The body contains the variable declarations and statements that are executed when the method is called
  - It will also return the calculated answer

```
public static double cubeVolume(double sideLength)
{
    double volume = sideLength * sideLength * sideLength;
    return volume;
}
```

# Back from the Box

---

- The values returned from `cubeVolume` are stored in local variables inside `main`
- The results are then printed out

```
public static void main(String[] args)
{
    double result1 = cubeVolume(2);
    double result2 = cubeVolume(10);
    System.out.println("A cube of side length 2 has volume
        " + result1);
    System.out.println("A cube of side length 10 has volume
        " + result2);
}
```

## Syntax 5.1 Method Declaration

**Syntax**    `public static returnType methodName(parameterType parameterName, . . . )`  
              `{`  
                  *method body*  
              `}`

Diagram illustrating the syntax of a Java method declaration with annotations:

```
public static double cubeVolume(double sideLength)
{
    double volume = sideLength * sideLength * sideLength;
    return volume;
}
```

Annotations:

- Type of return value:** `double`
- Name of method:** `cubeVolume`
- Type of parameter variable:** `double`
- Name of parameter variable:** `sideLength`
- Method body, executed when method is called:** The block between `{` and `}`.
- return statement exits method and returns result.** (Callout for the `return` statement)

# Cubes.java

---

```
1  /**
2   * This program computes the volumes of two cubes.
3   */
4  public class Cubes
5  {
6      public static void main(String[] args)
7      {
8          double result1 = cubeVolume(2);
9          double result2 = cubeVolume(10);
10         System.out.println("A cube with side length 2 has volume " + result1);
11         System.out.println("A cube with side length 10 has volume " + result2);
12     }
13
14     /**
15      * Computes the volume of a cube.
16      * @param sideLength the side length of the cube
17      * @return the volume
18      */
19     public static double cubeVolume(double sideLength)
20     {
21         double volume = sideLength * sideLength * sideLength;
22         return volume;
23     }
24 }
```

## Program Run

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```



## Self Check 5.5

---

What is the value of `cubeVolume(3)` ?

**Answer:** 27

## Self Check 5.6

---

What is the value of `cubeVolume(cubeVolume(2))` ?

**Answer:**  $8 \times 8 \times 8 = 512$

## Self Check 5.7

---

Provide an alternate implementation of the body of the `cubeVolume` method by calling the `Math.pow` method.

**Answer:**

```
double volume = Math.pow(sideLength, 3);  
return volume;
```

## Self Check 5.8

---

Declare a method `squareArea` that computes the area of a square of a given side length.

**Answer:**

```
public static double squareArea(double sideLength)
{
    double area = sideLength * sideLength;
    return area;
}
```

## Self Check 5.9

---

Consider this method:

```
public static int mystery(int x, int y)
{
    double result = (x + y) / (y - x);
    return result;
}
```

What is the result of the call `mystery(2, 3)`?

**Answer:**  $(2 + 3) / (3 - 2) = 5$

# Method Comments

---

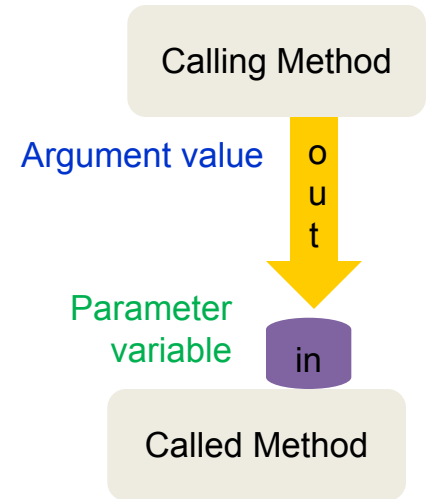
- Write a Javadoc comment above each method
- Start with `/**`
  - Note the purpose of the method
  - `@param` Describe each parameter variable
  - `@return` Describe the return value
- End with `*/`

```
/**  
    Computes the volume of a cube.  
    @param sideLength the side length of the cube  
    @return the volume  
*/  
public static double cubeVolume(double sideLength)
```

# Parameter Passing

---

- **Parameter variables** receive the **argument values** supplied in the method call
  - They both must be the same type
- The **argument value** may be:
  - The contents of a variable
  - A 'literal' value (2)
  - aka. 'actual parameter' or argument
- The **parameter variable** is:
  - Declared in the called method
  - Initialized with the value of the **argument value**
  - Used as a variable inside the called method
  - aka. 'formal parameter'



# Parameter Passing Steps

---

```
public static void main(String[] args)
{
    double result1 = cubeVolume(2);
    . . .
}
```

result1 = 8

```
public static double cubeVolume(double sideLength)
{
    double volume = sideLength * sideLength * sideLength;
    return volume;
}
```

sideLength = 2

volume = 8



## Self Check 5.10

---

What does this program print? Use a diagram like Figure 3 to find the answer.

```
public static double mystery(int x, int y)
{
    double z = x + y;
    z = z / 2.0;
    return z;
}
public static void main(String[] args)
{
    int a = 5;
    int b = 7;
    System.out.println(mystery(a, b));
}
```

**Answer:** When the `mystery` method is called, `x` is set to 5, `y` is set to 7, and `z` becomes 12.0. Then `z` is changed to 6.0, and that value is returned and printed.

## Self Check 5.11

---

What does this program print? Use a diagram like Figure 3 to find the answer.

```
public static int mystery(int x)
{
    int y = x * x;
    return y;
}
public static void main(String[] args)
{
    int a = 4;
    System.out.println(mystery(a + 1));
}
```

**Answer:** When the method is called, x is set to 5. Then y is set to 25, and that value is returned and printed.

## Self Check 5.12

---

What does this program print? Use a diagram like Figure 3 to find the answer.

```
public static int mystery(int n)
{
    n++;
    n++;
    return n;
}
public static void main(String[] args)
{
    int a = 5;
    System.out.println(mystery(a));
}
```

**Answer:** When the method is called, `n` is set to 5. Then `n` is incremented twice, setting it to 7. That value is returned and printed.

# Common Error

- Trying to Modify Arguments

- A copy of the argument values is passed

- Called method (`addTax`) can modify local copy (`price`)

- But not original

- in calling method

- `total`

```
public static void main(String[] args)
{
    double total = 10;
    addTax(total, 7.5);
}
```

10.0

copy

```
public static int addTax(double price, double rate)
{
    double tax = price * rate / 100;
    price = price + tax; // Has no effect outside the method
    return tax;
}
```

price

10.75

# Return Values

---

- Methods can (optionally) return one value
  - Declare a **return type** in the method declaration
  - Add a **return statement** that returns a value
    - A **return statement** does two things:
      - Immediately terminates the method
      - Passes the return value back to the calling method

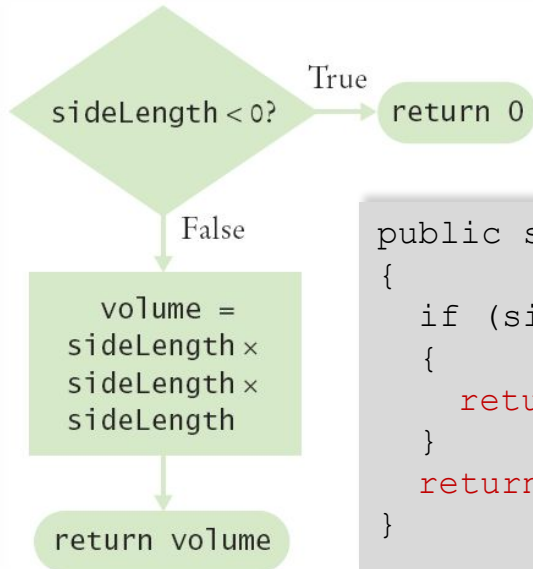
```
public static double cubeVolume (double sideLength)
{
    double volume = sideLength * sideLength * sideLength;
    return volume;
}
```

The diagram highlights the **return type** 'double' in the method signature and the **return statement** 'return volume;' within the method body.

- The return value may be a value, a variable or a calculation
  - Type must match return type

# Multiple return Statements

- A method can use multiple `return` statements
  - But every branch must have a `return` statement



```
public static double cubeVolume(double sideLength)
{
    if (sideLength < 0)
    {
        return 0;
    }
    return sideLength * sideLength * sideLength;
}
```

## Self Check 5.13

---

Suppose we change the body of the `cubeVolume` method to

```
if (sideLength <= 0) { return 0; }  
return sideLength * sideLength * sideLength;
```

How does this method differ from the one described in this section?

**Answer:** It acts the same way: If `sideLength` is 0, it returns 0 directly instead of computing  $0 \times 0 \times 0$ .

## Self Check 5.14

---

What does this method do?

```
public static boolean mystery(int n)
{
    if (n % 2 == 0) { return true; }
    else { return false; }
}
```

**Answer:** It returns `true` if `n` is even; `false` if `n` is odd.



## Self Check 5.15

---

Implement the `mystery` method of Self Check 14 with a single `return` statement.

**Answer:**

```
public static boolean mystery(int n)
{
    return n % 2 == 0;
}
```

# Common Error

---

- Missing `return` Statement

- Make sure all conditions are handled
- In this case, `x` could be equal to 0
  - No `return` statement for this condition
  - The compiler will complain if any branch has no `return` statement

```
public static int sign(double x)
{
    if (x < 0) { return -1; }
    if (x > 0) { return 1; }
    // Error: missing return value if x equals 0
}
```

# Methods without Return Values

---

- Methods are not required to return a value
  - The return type of `void` means nothing is returned
  - No `return` statement is required, but a `return` without a value can be coded
  - The method can generate output though!

```
...  
boxString("Hello");  
...
```

```
-----  
!Hello!  
-----
```

```
public static void boxString(String str)  
{  
    int n = str.length();  
    for (int i = 0; i < n + 2; i++)  
        { System.out.print("-"); }  
    System.out.println();  
    System.out.println("!" + str + "!");  
    for (int i = 0; i < n + 2; i++)  
        { System.out.print("-"); }  
    System.out.println();  
}
```

# Using return Without a Value

---

- You can use the `return` statement without a value
  - In methods with `void` return type
  - The method will terminate immediately!

```
public static void boxString(String str)
{
    int n = str.length();
    if (n == 0)
    {
        return; // Return immediately
    }
    for (int i = 0; i < n + 2; i++) { System.out.print("-"); }
    System.out.println();
    System.out.println("!" + str + "!");
    for (int i = 0; i < n + 2; i++) { System.out.print("-"); }
    System.out.println();
}
```

## Self Check 5.16

---

How do you generate the following printout, using the `boxString` method?

```
-----  
!Hello!  
-----  
  
-----  
!World!  
-----
```

**Answer:**

```
boxString("Hello");  
boxString("World");
```

## Self Check 5.17

---

What is wrong with the following statement?

```
System.out.print(boxString("Hello"));
```

**Answer:** The `boxString` method does not return a value. Therefore, you cannot use it in a call to the `print` method.

## Self Check 5.18

---

Implement a method `shout` that prints a line consisting of a string followed by three exclamation marks. For example, `shout("Hello")` should print `Hello!!!`. The method should not return a value.

### Answer:

```
public static void shout(String message)
{
    System.out.println(message + "!!!");
}
```

## Self Check 5.19

---

How would you modify the `boxString` method to leave a space around the string that is being boxed, like this:

```
-----  
! Hello !  
-----
```

### Answer:

```
public static void boxString(String contents)
{
    int n = contents.length();
    for (int i = 0; i < n + 4; i++)
    {
        System.out.print("-");
    }
    System.out.println();
    System.out.println("! " + contents + " !");
    for (int i = 0; i < n + 4; i++)
    {
        System.out.print("-");
    }
    System.out.println()
}
```



## Self Check 5.20

---

The `boxString` method contains the code for printing a line of - characters twice. Place that code into a separate method `printLine`, and use that method to simplify `boxString`. What is the code of both methods?

### Answer:

```
public static void printLine(int count)
{
    for (int i = 0; i < count; i++)
    {
        System.out.print("-");
    }
    System.out.println();
}
public static void boxString(String contents)
{
    int n = contents.length();
    printLine(n + 2);
    System.out.println("!" + contents + "!");
    printLine(n + 2);
}
```

# Problem Solving: Reusable Methods

---

- Find Repetitive Code
  - May have different values but same logic

```
int hours;
do
{
    System.out.print("Enter a value between 1 and 12: ");
    hours = in.nextInt();
}
while (hours < 1 || hours > 12);

int minutes;
do
{
    System.out.print("Enter a value between 0 and 59: ");
    minutes = in.nextInt();
}
while (minutes < 0 || minutes > 59);
```



1 - 12


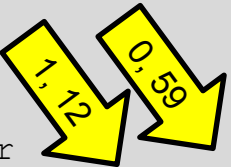


0 - 59

# Write a 'Parameterized' Method

---

```
/**
 * Prompts a user to enter a value in a given range until the user
 * provides a valid input.
 * @param low the low end of the range
 * @param high the high end of the range
 * @return the value provided by the user
 */
public static int readValueBetween(int low, int high)
{
    int input;
    do
    {
        System.out.print("Enter between " + low + " and " + high + ": ");
        Scanner in = new Scanner(System.in);
        input = in.nextInt();
    }
    while (input < low || input > high);
    return input;
}
```



## Self Check 5.21

---

Consider the following statements:

```
int totalPennies = (int) Math.round(100 * total) % 100;
int taxPennies = (int) Math.round(100 * (total * taxRate)) %
100;
```

Introduce a method to reduce code duplication.

**Answer:**

```
int totalPennies = getPennies(total);
int taxPennies = getPennies(total * taxRate);
where the method is defined as
/**
    Gets a given amount in pennies.
    @param amount an amount in dollars and cents
    @return the number of pennies in the amount
 */
public static int getPennies(double amount)
{
    return (int) Math.round(100 * amount) % 100;
}
```

## Self Check 5.22

---

Consider this method that prints a page number on the left or right side of a page:

```
if (page % 2 == 0) { System.out.println(page); }  
else { System.out.println("                " + page); }  
}
```

Introduce a method with return type `boolean` to make the condition in the `if` statement easier to understand.

### Answer:

```
if (isEven(page)) . . .
```

where the method is defined as follows:

```
public static boolean isEven(int n)  
{  
    return n % 2 == 0;  
}
```

## Self Check 5.23

---

Consider the following method that computes compound interest for an account with an initial balance of \$10,000 and an interest rate of 5 percent:

```
public static double balance(int years) { return 10000 *  
Math.pow(1.05, years); }
```

How can you make this method more reusable?

**Answer:** Add parameter variables so you can pass the initial balance and interest rate to the method:

```
public static double balance(  
    double initialBalance, double rate,  
    int years)  
{  
    return initialBalance * pow(1 + rate / 100, years);  
}
```

## Self Check 5.24

---

The comment explains what the following loop does. Use a method instead.

```
// Counts the number of spaces
int spaces = 0;
for (int i = 0; i < input.length(); i++)
{
    if (input.charAt(i) == ' ') { spaces++; }
}
```

### Answer:

```
int spaces = countSpaces(input);
where the method is defined as follows:
/**
    Gets the number of spaces in a string.
    @param str any string
    @return the number of spaces in str
*/
public static int countSpaces(String str)
{
    int count = 0;
    for (int i = 0; i < str.length(); i++)
    {
        if (str.charAt(i) == ' ')
        {
            count++;
        }
    }
    return count;
}
```

## Self Check 5.25

---

In Self Check 24, you were asked to implement a method that counts spaces. How can you generalize it so that it can count any character? Why would you want to do this?

**Answer:** It is very easy to replace the space with any character.

```
/**
 * Counts the instances of a given character in a string.
 * @param str any string
 * @param ch a character whose occurrences should be counted
 * @return the number of times that ch occurs in str
 */
public static int count(String str, char ch)
{
    int count = 0;
    for (int i = 0; i < str.length(); i++)
    {
        if (str.charAt(i) == ch) { count++; }
    }
    return count;
}
```

This is useful if you want to count other characters. For example, `count(input, ",", ")` counts the commas in the input.

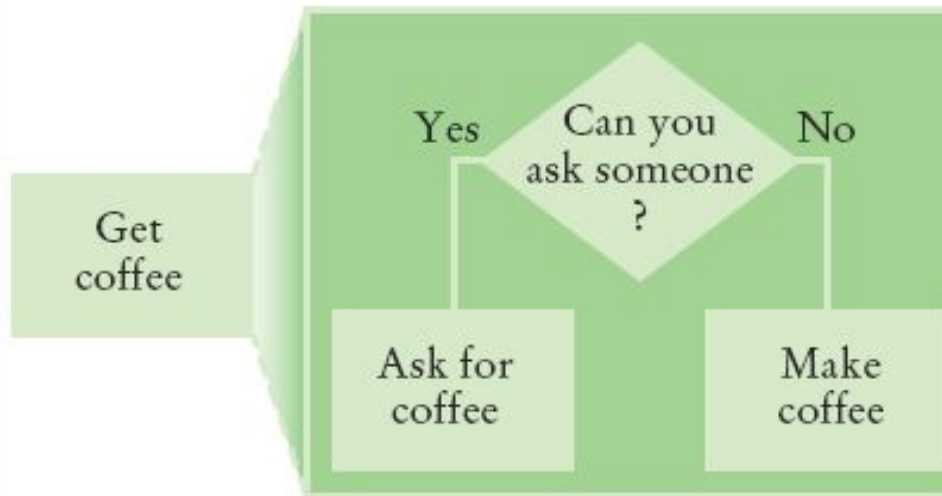


# Problem Solving

---

- Stepwise Refinement

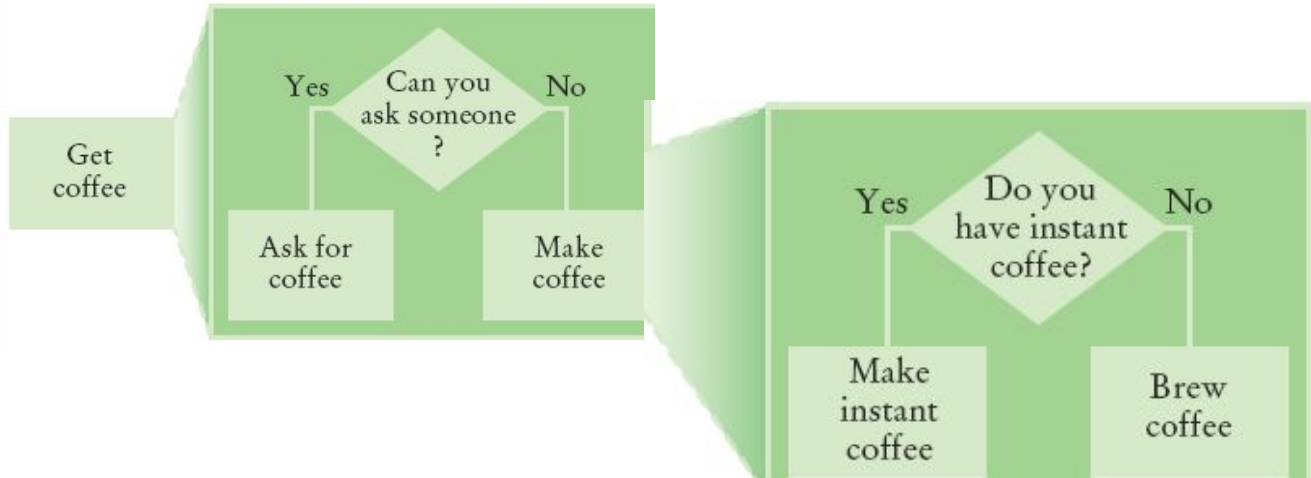
- To solve a difficult task, break it down into simpler tasks
- Then keep breaking down the simpler tasks into even simpler ones, until you are left with tasks that you know how to solve



# Get Coffee

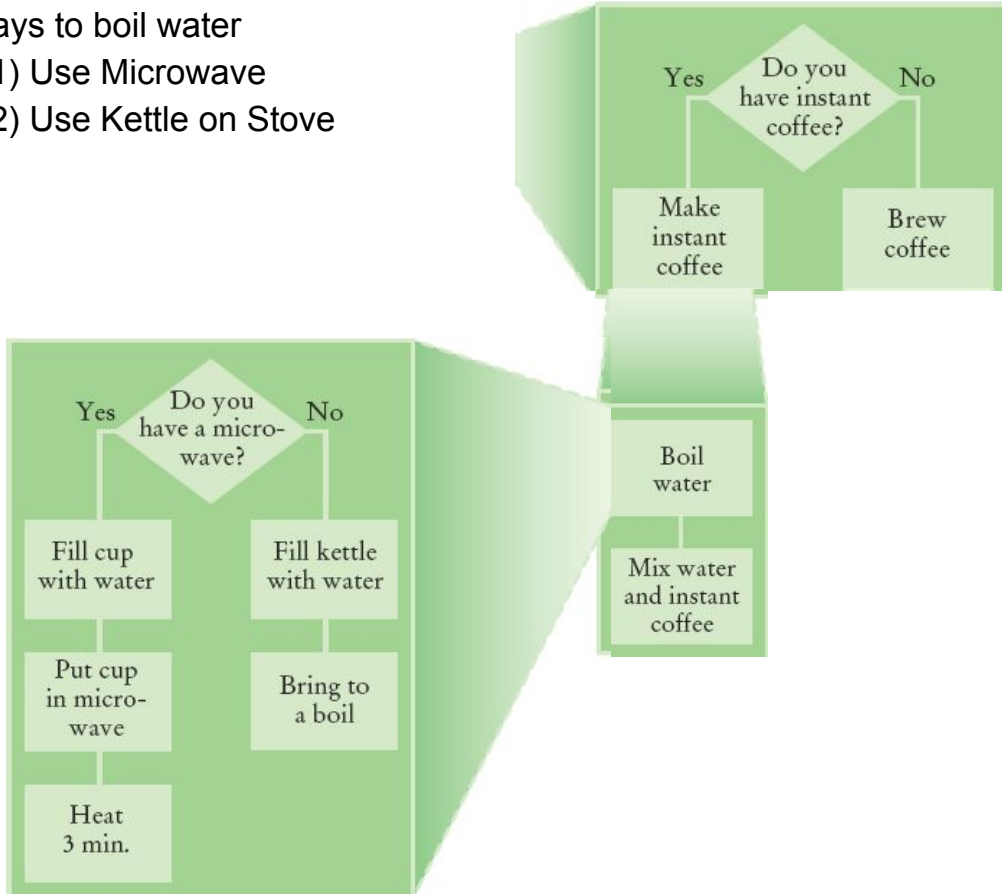
---

- If you must make coffee, there are two ways:
  - Make Instant Coffee
  - Brew Coffee



# Instant Coffee

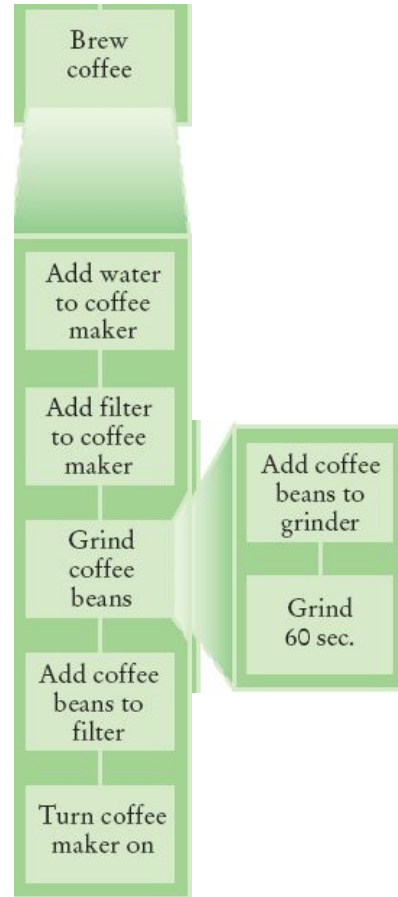
- Two ways to boil water
  - 1) Use Microwave
  - 2) Use Kettle on Stove



# Brew Coffee

---

- Assumes coffee maker
  - Add water
  - Add filter
  - Grind Coffee
    - Add beans to grinder
    - Grind 60 seconds
  - Fill filter with ground coffee
  - Turn coffee maker on
- Steps are easily done



# Stepwise Refinement Example

- When printing a check, it is customary to write the check amount both as a number (“\$274.15”) and as a text string (“two hundred seventy four dollars and 15 cents”). Write a program to turn a number into a text string.
- Wow, sounds difficult!
- Break it down
  - Let’s take the dollar part (274) and come up with a plan
  - Take an Integer from 0 – 999
  - Return a String
  - Still pretty hard...



# Stepwise Refinement Example

---

- Take it digit by digit (2, 7, 4) – left to right
- Handle the first digit (hundreds)
  - If empty, we're done with hundreds
  - Get first digit (Integer from 1 – 9)
  - Get digit name (“one”, “two”, “three”...)
  - Add the word “hundred”
  - Sounds easy!
- Second digit (tens)
  - Get second digit (Integer from 0 – 9)
  - If 0, we are done with tens... handle third digit
  - If 1, ... may be eleven, twelve.. Teens... Not easy!
    - Let's look at each possibility left (1x-9x)...

# Stepwise Refinement Example

---

- If second digit is a 0
  - Get third digit (Integer from 0 – 9)
  - Get digit name (“”, “one”, “two”...) ... Same as before?
  - Sounds easy!
- If second digit is a 1
  - Get third digit (Integer from 0 – 9)
  - Return a String (“ten”, “eleven”, “twelve”...)
- If second digit is a 2-9
  - Start with string “twenty”, “thirty”, “forty” ...
  - Get third digit (Integer from 0 – 9)
  - Get digit name (“”, “one”, “two”...) ... Same as before
  - Sounds easy!

# Name the Sub-Tasks

---

- `digitName`
  - Takes an Integer from 0 – 9
  - Return a String (“”, “one”, “two”...)
- `tensName` (second digit  $\geq 20$ )
  - Takes an Integer from 0 – 9
  - Return a String (“twenty”, “thirty”...) plus
    - `digitName` (third digit)
- `teenName`
  - Takes an Integer from 0 – 9
  - Return a String (“ten”, “eleven”...)



# Write Pseudocode

---

part = number (The part that still needs to be converted)

name = "" (The name of the number)

If part  $\geq$  100

    name = name of hundreds in part + " hundred"

    Remove hundreds from part.

If part  $\geq$  20

    Append tensName(part) to name.

    Remove tens from part.

Else if part  $\geq$  10

    Append teenName(part) to name.

    part = 0

If (part  $>$  0)

    Append digitName(part) to name.

# Plan The Methods

---

- Decide on name, parameter(s) and types and return type
- String `digitName(int number)`
  - Return a String (“”, “one”, “two”...)
- String `tensName(int number)`
  - Return a String (“twenty”, “thirty”...) plus
    - Return from `digitName(thirdDigit)`
- String `teenName(int number)`
  - Return a String (“ten”, “eleven”...)

# Convert to Java: intName method

---

- main calls intName

- Does all the work
- Returns a String

- Uses methods:

- tensName
- teenName
- digitName

```
21 public static String intName(int number)
22 {
23     int part = number; // The part that still needs to be converted
24     String name = ""; // The name of the number
25
26     if (part >= 100)
27     {
28         name = digitName(part / 100) + " hundred";
29         part = part % 100;
30
31
32     if (part >= 20)
33     {
34         name = name + " " + tensName(part);
35         part = part % 10;
36     }
37     else if (part >= 10)
38     {
39         name = name + " " + teenName(part);
40         part = 0;
41     }
42
43     if (part > 0)
44     {
45         name = name + " " + digitName(part);
46     }
47
48     return name;
49 }
```

## digitName, teenName, tensName

```
56 public static String digitName(int digit)
57 {
58     if (digit == 0) { return "zero"; }
59     if (digit == 1) { return "one"; }
60     if (digit == 2) { return "two"; }
61     if (digit == 3) { return "three"; }
62     if (digit == 4) { return "four"; }
63     if (digit == 5) { return "five"; }
64     if (digit == 6) { return "six"; }
65     if (digit == 7) { return "seven"; }
66     if (digit == 8) { return "eight"; }
67     if (digit == 9) { return "nine"; }
68 }

    public static String teenName(int number)
    {
        if (number == 10) { return "ten"; }
        if (number == 11) { return "eleven"; }
        if (number == 12) { return "twelve"; }
        if (number == 13) { return "thirteen"; }
        if (number == 14) { return "fourteen"; }
        if (number == 15) { return "fifteen"; }
        if (number == 16) { return "sixteen"; }
        if (number == 17) { return "seventeen"; }
        if (number == 18) { return "eighteen"; }
        if (number == 19) { return "nineteen"; }
    }

    public static String tensName(int number)
    {
        if (number >= 90) { return "ninety"; }
        if (number >= 80) { return "eighty"; }
        if (number >= 70) { return "seventy"; }
        if (number >= 60) { return "sixty"; }
        if (number >= 50) { return "fifty"; }
        if (number >= 40) { return "forty"; }
        if (number >= 30) { return "thirty"; }
        if (number >= 20) { return "twenty"; }
        return "";
    }
}
```

### Program Run

Please enter a positive integer: 729  
seven hundred twenty nine

## Self Check 5.26

---

Explain how you can improve the `intName` method so that it can handle arguments up to 9999.

**Answer:** Change line 28 to

```
name = name + digitName(part / 100)
+ " hundred";
```

In line 25, add the statement

```
if (part >= 1000)
{
    name = digitName(part / 1000) + "thousand ";
    part = part % 1000;
}
```

In line 18, change 1,000 to 10,000 in the comment.

## Self Check 5.27

---

Why does line 40 set `part = 0`?

**Answer:** In the case of “teens”, we already have the last digit as part of the name.

## Self Check 5.28

---

What happens when you call `intName(0)` ? How can you change the `intName` method to handle this case correctly?

**Answer:** Nothing is printed. One way of dealing with this case is to add the following statement before line 23.

```
if (number == 0) { return "zero"; }
```

## Self Check 5.29

---

Trace the method call `intName(72)`, as described in Programming Tip 5.4.

**Answer:** Here is the approximate trace:

intName(number = 72)	
part	name
<del>72</del>	<del>"seventy"</del>
2	"seventy two"

Note that the string starts with a blank space. Exercise P5.5 asks you to eliminate it.



## Self Check 5.30

---

Use the process of stepwise refinement to break down the task of printing the following table into simpler tasks.

i		i * i	
1		1	
2		8	
20		8000	

**Answer:** Here is one possible solution. Break up the task print table into print header and print body. The print header task calls print separator, prints the header cells, and calls print separator again. The print body task repeatedly calls print row and then calls print separator.

# Programming Tips

- Keep methods short
  - If more than one screen, break into 'sub' methods
- Trace your methods
  - One line for each step
  - Columns for key variables
- Use Stubs as you write larger programs
  - Unfinished methods that return a 'dummy' value

intName(number = 416)	
part	name
<del>416</del>	<del>m</del>
<del>16</del>	<del>"four hundred"</del>
0	"four hundred sixteen"

```
public static String digitName(int digit)
{
    return "mumble";
}
```

# Variable Scope

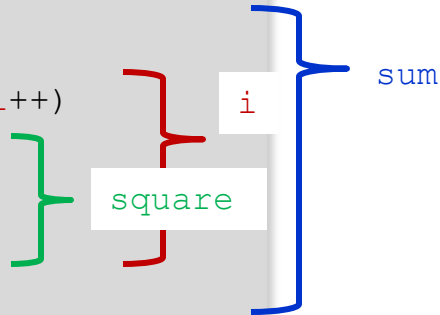
---

- Variables can be declared:
  - Inside a method
    - Known as 'local variables'
    - Only available inside this method
    - Parameter variables are like local variables
  - Inside a block of code {     }
  - Sometimes called 'block scope'
  - If declared inside block { ends at end of block }
  - Outside of a method
    - Sometimes called 'global scope'
    - Can be used (and changed) by code in any method
- How do you choose?

# Examples of Scope

- `sum` is a local variable in `main`
- `square` is only visible inside the `for` loop block
- `i` is only visible inside the `for` loop

```
public static void main(String[] args)
{
    int sum = 0;
    for (int i = 1; i <= 10; i++)
    {
        int square = i * i;
        sum = sum + square;
    }
    System.out.println(sum);
}
```



The scope of a variable is the part of the program in which it is visible.

# Local Variables of Methods

---

- Variables declared inside one method are not visible to other methods
  - `sideLength` is local to `main`
  - Using it outside `main` will cause a compiler error

```
public static void main(String[] args)
{
    double sideLength = 10;
    int result = cubeVolume();
    System.out.println(result);
}

public static double cubeVolume()
{
    return sideLength * sideLength * sideLength; // ERROR
}
```

# Re-using Names for Local Variables

---

- Variables declared inside one method are not visible to other methods
  - `result` is local to `square` and `result` is local to `main`
  - They are two different variables and do not overlap

```
public static int square(int n)
{
    int result = n * n;
    return result;
}
```



`result`

```
public static void main(String[] args)
{
    int result = square(3) + square(4);
    System.out.println(result);
}
```

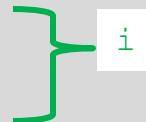
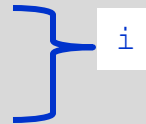


`result`

# Re-using Names for Block Variables

- Variables declared inside one block are not visible to other methods
  - `i` is inside the first `for` block and `i` is inside the second
  - They are two different variables and do not overlap

```
public static void main(String[] args)
{
    int sum = 0;
    for (int i = 1; i <= 10; i++)
    {
        sum = sum + i;
    }
    for (int i = 1; i <= 10; i++)
    {
        sum = sum + i * i;
    }
    System.out.println(sum);
}
```



# Overlapping Scope

- Variables (including parameter variables) must have unique names within their scope
  - `n` has local scope and `n` is in a block inside that scope
  - The compiler will complain when the block scope `n` is declared

```
public static int sumOfSquares(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; i++)
    {
        int n = i * i; // ERROR
        sum = sum + n;
    }
    return sum;
}
```

Local n

block scope n



# Global and Local Overlapping

- Global and Local (method) variables can overlap
  - The local `same` will be used when it is in scope
  - No access to global `same` when local `same` is in scope

```
public class Scoper
{
    public static int same;    // 'global'
    public static void main(String[] args)
    {
        int same = 0;        // local
        for (int i = 1; i <= 10; i++)
        {
            int square = i * i;
            same = same + square;
        }
        System.out.println(same);
    }
}
```

same

same

Variables in different scopes with the same name will compile, but it is not a good idea

# Self Check

---

Consider this sample program, then answer the questions below.

```
public class Sample
{
    public static void main(String[] args)
    {
        int x = 4;
        x = mystery(x + 1);
        System.out.println(s);
    }

    public static int mystery(int x)
    {
        int s = 0;
        for (int i = 0; i < x; x++)
        {
            int x = i + 1;
            s = s + x;
        }
        return s;
    }
}
```

## Self Check 5.31

---

Which lines are in the scope of the variable `i` declared in line 13?

**Answer:** Lines 14-17.

## Self Check 5.32

---

Which lines are in the scope of the parameter variable  $x$  declared in line 10?

**Answer:** Lines 11-19.

## Self Check 5.33

---

The program declares two local variables with the same name whose scopes don't overlap. What are they?

**Answer:** The variables `x` defined in lines 5 and 15.

## Self Check 5.34

---

There is a scope error in the mystery method. How do you fix it?

**Answer:** Rename the local variable `x` that is declared in line 15, or rename the parameter variable `x` that is declared in line 10.

## Self Check 5.35

---

There is a scope error in the `main` method. What is it, and how do you fix it?

**Answer:** The `main` method accesses the local variable `s` of the `mystery` method. Assuming that the `main` method intended to print the last value of `s` before the method returned, it should simply print the return value that is stored in its local variable `x`.

## Recursive Methods (optional)

---

- A recursive method is a method that calls itself
- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
- For a recursion to terminate, there must be special cases for the simplest inputs



# Recursive Triangle Example

---

- The method will call itself (and not output anything) until `sideLength` becomes `< 1`
- It will then use the return statement and each of the previous iterations will print their results
  - 1, 2, 3 then 4

```
public static void printTriangle(int sideLength)
{
    if (sideLength < 1) { return; }

    printTriangle(sideLength - 1);
    for (int i = 0; i < sideLength; i++)
    {
        System.out.print("[]");
    }
    System.out.println();
}
```

```
[]
>[] []
>[] [] []
>[] [] [] []
```

Print the triangle with side length 3.

Print a line with four [].

## Recursive Calls and Returns

---

Here is what happens when we print a triangle with side length 4.

- The call `printTriangle(4)` calls `printTriangle(3)`.
  - The call `printTriangle(3)` calls `printTriangle(2)`.
    - The call `printTriangle(2)` calls `printTriangle(1)`.
      - The call `printTriangle(1)` calls `printTriangle(0)`.
        - The call `printTriangle(0)` returns, doing nothing.
      - The call `printTriangle(1)` prints `[]`.
    - The call `printTriangle(2)` prints `[] []`.
  - The call `printTriangle(3)` prints `[] [] []`.
- The call `printTriangle(4)` prints `[] [] [] []`.

## Self Check 5.36

---

Consider this slight modification of the `printTriangle` method:

```
public static void printTriangle(int sideLength)
{
    if (sideLength < 1) { return; }
    for (int i = 0; i < sideLength; i++)
    {
        System.out.print("[ ]");
    }
    System.out.println();
    printTriangle(sideLength - 1);
}
```

What is the result of `printTriangle(4)` ?

**Answer:**

```
[ ] [ ] [ ] [ ]
[ ] [ ] [ ]
[ ] [ ]
[ ]
```

## Self Check 5.37

---

Consider this recursive method:

```
public static int mystery(int n)
{
    if (n <= 0) { return 0; }
    return n + mystery(n - 1);
}
```

What is `mystery(4)` ?

**Answer:**  $4 + 3 + 2 + 1 + 0 = 10$

## Self Check 5.38

---

Consider this recursive method:

```
public static int mystery(int n)
{
    if (n <= 0) { return 0; }
    return mystery(n / 2) + 1;
}
```

What is `mystery(20)` ?

**Answer:**  $\text{mystery}(10) + 1 = \text{mystery}(5) + 2 = \text{mystery}(2) + 3 = \text{mystery}(1) + 4 = \text{mystery}(0) + 5 = 5$

## Self Check 5.39

---

Write a recursive method for printing  $n$  box shapes `[]` in a row.

**Answer:** The idea is to print one `[]`, then print  $n - 1$  of them.

```
public static void printBoxes(int n)
{
    if (n == 0) { return; }
    System.out.print("[]");
    printBoxes(n - 1);
}
```

## Self Check 5.40

---

The `intName` method in Section 5.7 accepted arguments  $< 1,000$ . Using a recursive call, extend its range to 999,999. For example an input of 12,345 should return "twelve thousand three hundred forty five".

**Answer:** Simply add the following to the beginning of the method:

```
if (part >= 1000)
{
    return intName(part / 1000) + " thousand "
        + intName(part % 1000);
}
```