Software and Programming II

Methods

Keith Mannock and Oded Lachish
based on materials by Cay Horstmann and an original slide deck
by Donald W. Smith

Department of Computer Science and Information Systems
Birkbeck, University of London

keith@dcs.bbk.ac.uk and oded@dcs.bbk.ac.uk

September 30, 2014

Please Note:

These *slides* also act as “notes” to remind you of the topics you should be familiar with.

Objectives

- Refresher on certain topics
- 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

Contents

1. Methods as Black Boxes
2. Implementing Methods
3. Parameter Passing
4. Return Values
5. Problem Solving via Stepwise refinement
6. Variable Scope
7. Recursive Methods

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

- You call a method in order to execute it's instructions

What is a method?

- Some methods you have already used are, for example:
  - `Math.pow()`
  - `String.length()`
  - `Character.isDigit()`
  - `Scanner.nextInt()`
  - `main()`
- They:
  - may have a capitalized name and a dot (`.`) before them
  - a method name

    Follow the same rules as variable names, *camelCase* style
  - ( ) — a set of parenthesis at the end

    A place to provide the method input information

Flowchart of Calling a Method

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

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

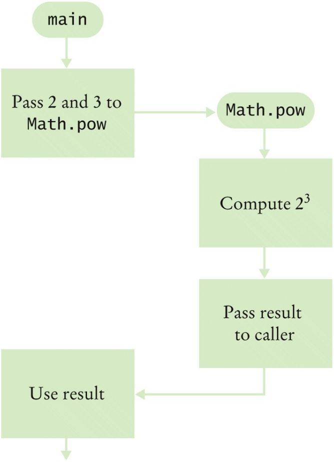


Figure 5.1
© John Wiley & Sons, Inc. All rights reserved.

Arguments and Return Values

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

- `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`

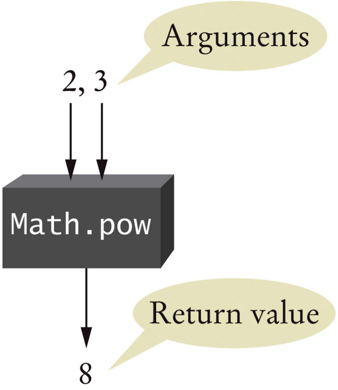


Figure 5.2
© John Wiley & Sons, Inc. All rights reserved.

Black Box Analogy

- A thermostat is a “black box”
  - Set a desired temperature
  - Turns on heater/AC as required
  - You dont have to know how it really works!
- Use methods like black boxes
  - Pass the method what it needs to do its job
  - Receive the answer

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 writing this method:
    - 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`
    - Note the difference between formal and actual parameters

```
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);
}
```

Method Comments

- Write a `Javadoc` comment above each method
- Start with /**
  - State 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)
```

Do not try and modify arguments!

- A copy of the argument values is passed
- Called method (addTax) can modify local copy (price)
- But not the original in the calling method (total)

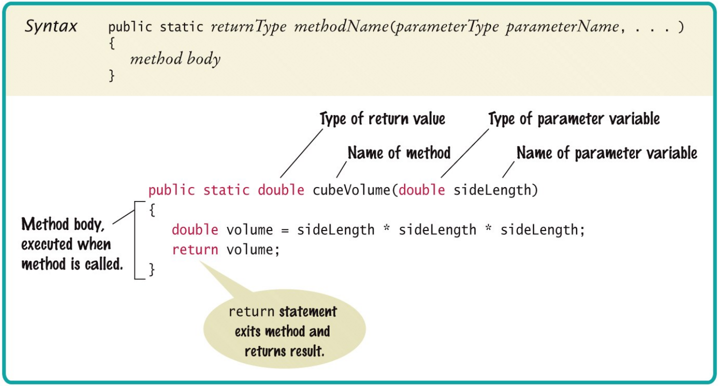
```
public static void main(String[] args) {
  double total = 10;
  addTax(total, 7.5);
}

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

Return Values I

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:
    1. Immediately terminates the method
    2. Passes the return value back to the calling method
- The return value may be a value, a variable or a calculation
- Type must match return type

Return Values II



Syntax

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

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

Type of return value
Type of parameter variable
Name of method
Name of parameter variable
Method body, executed when method is called.
return statement exits method and returns result.

Syntax 5.1
© John Wiley & Sons, Inc. All rights reserved.

Multiple `return` statements

- A method can use multiple return statements
- Every branch must have a return statement

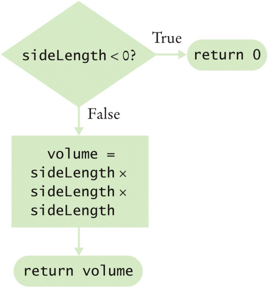


Figure 5.4b
© John Wiley & Sons, Inc. All rights reserved.

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

Missing `return` statement

- Make sure all conditions are handled
- In the following 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
- The method can generate output though!
- Other side effects, for example, assignment, are less desirable

```
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

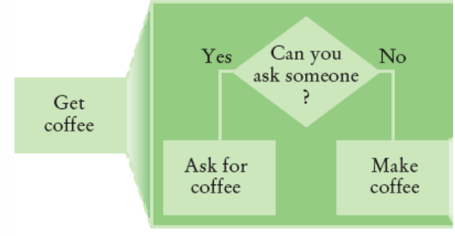
You can use the return statement without a value

- In methods with void return type
- The method will terminate immediately!

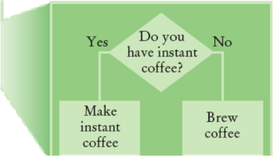
```
public static void boxString(String str) {
  int n = str.length();
  if (n == 0) {
    return; // Return immediately
  }
  for (int i = 0; i < n + 2; i++) { System.out.print("-"); }
  System.out.println();
  System.out.println("!" + str + "!");
  for (int i = 0; i < n + 2; i++) { System.out.print("-");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

Problem Solving: Stepwise refinement (I)

1. To solve a difficult task, break it down into simpler tasks
2. Then keep breaking down the simpler tasks into even simpler ones
3. Until you are left with tasks that you know how to solve



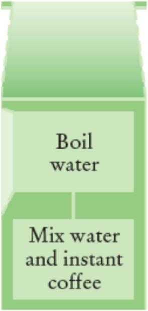
Problem Solving: Stepwise refinement (II)

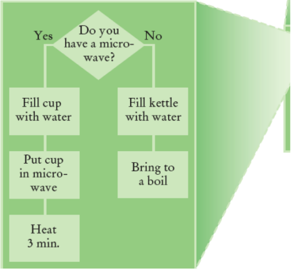


If you must make coffee, there are two ways:

1. Make Instant Coffee
2. Brew Coffee

Problem Solving: Stepwise refinement (III)

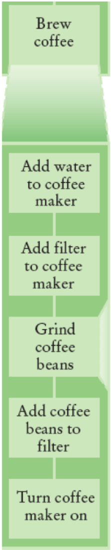




Two ways to boil water

1. Use Microwave
2. Use Kettle on Stove

Problem Solving: Stepwise refinement (IV)


Brew coffee
Add water to coffee maker
Add filter to coffee maker
Grind coffee beans
Add coffee beans to filter
Turn coffee maker on

Brew Coffee — Assumes coffee maker

- Add water
- Add filter
- Grind Coffee
  - Add beans to grinder
  - Grind for 60 seconds
- Fill filter with ground coffee
- Turn coffee maker on

Individual steps are easily completed

Example

When printing a cheque, it is customary to write the cheque amount both as a number ("£274.15") and as a text string ("two hundred seventy four pounds and 15 pence).

Write a program to turn a number into a text string.

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

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

Variable scope

Variables can be declared:

- Inside a method
  - Known as local variables
  - Only available inside the 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);
}
```

Local variables of methods

Variables declared inside one method are not visible to other methods

- `sideLength` is *local* to `main`
- Using the variable 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 the method `square` and a different `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;
}

public static void main(String[] args){
  int result = square(3) + square(4);
  System.out.println(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 a different `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, the formal parameter, has local scope and the second 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;
}
```

Global and local overlapping scope

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);
  }
}
```

And now we consider *recursion* — a major technique in modern programming

Recursive Methods

- 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

An example. . . I

Examine this code carefully:

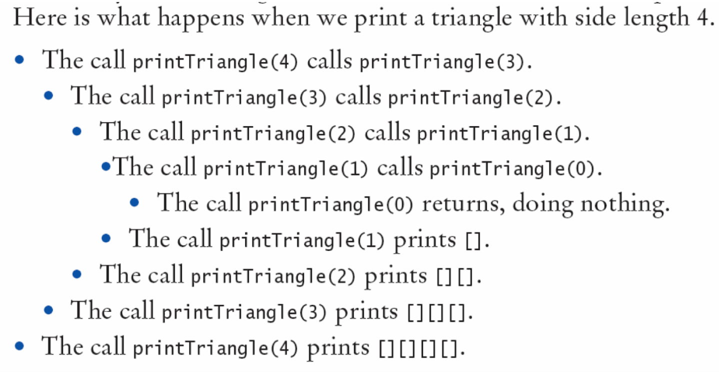
```
public static void printTriangle(int sideLength){
  if (sideLength < 1) { return; }

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

An example. . . II

- 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

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

Recursive calls and the 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 `[][][][]`.

Example: Triangle Numbers

- Will use recursion to compute the area of a triangle of width n , assuming each `[]` square has an area of 1
- Also called the n^{th} triangle number
- The third triangle number is 6, the fourth is 10

Outline of a `Triangle` class

```
public class Triangle {
  private int width;

  public void setWidth(int aWidth){
       width = aWidth;
  }

  public int getArea() {
    ...
  }
}
```

Handling a triangle of width 1

- The triangle consists of a single square
- It's area is 1
- Take care of this case first:

```
public int getArea() {
  if (width == 1) {
    return 1;
  }
  ...
}
```

Handling the general case

- Assume we know the area of the smaller triangle:

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

- Area of larger triangle can be calculated as

```
smallerArea + width
```

- To get the area of the smaller triangle make a smaller triangle and ask it for it's area:

```
Triangle smallerTriangle = new Triangle();
smallerTriangle.setWidth(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Completed getArea method

```
public int getArea() {
   if (width == 1) return 1;
   Triangle smallerTriangle = new Triangle();
   smallerTriangle.setWidth(width - 1);
   int smallerArea = smallerTriangle.getArea();
   return smallerArea + width;
}
```

Computing the area of a triangle with width 4

- getArea method makes a smaller triangle of width 3
- It calls getArea on that triangle
    - That method makes a smaller triangle of width 2
    - It calls getArea on that triangle
        - That method makes a smaller triangle of width 1
        - It calls getArea on that triangle
        - That method returns 1
        - The method returns smallerArea + width $= 1 + 2 = 3$
    - The method returns smallerArea + width $= 3 + 3 = 6$
- The method returns smallerArea + width $= 6 + 4 = 10$

Recursive Computation

- A recursive computation solves a problem by using the solution to the same problem with simpler inputs
- Call pattern of a recursive method is complicated
- So dont think about it — just do it!
- Every recursive call must simplify the computation in some way
- There must be special cases to handle the simplest computations directly

Example: Palindrome

- We wish to test whether a sentence is a *palindrome*
- A Palindrome is a string that is equal to itself when you reverse all the characters (ignoring the punctuation)

```
A man, a plan, a canal  Panama!
Go hang a salami, Im a lasagna hog
Madam, I'm Adam
```

Implementation of a isPalindrome method (I)

```
/**
    Tests whether a text is a palindrome.
    @param text a string that is being checked
    @return true if text is a palindrome, false otherwise
*/
public static boolean isPalindrome(String Text) {
    . . .
}
```

Implementation of a `isPalindrome` method (II)

Consider various ways to simplify inputs of which there are several possibilities:

- Remove the first character
- Remove the last character
- Remove both the first and last characters
- Remove a character from the middle
- Cut the character string into two halves
- . . .

Implementation of a `isPalindrome` method (III)

- Combine solutions with simpler inputs into a solution of the original problem
- Most promising simplification: *remove both first and last characters*

  ``‘‘adam, Im Ada’’ is a palindrome too``
- Thus, a word is a palindrome if
  - The first and last letters match, and
  - The word obtained by removing the first and last letters is also a palindrome

Implementation of a `isPalindrome` method (IV)

What if first or last character is not a letter? Ignore it

If the first and last characters are letters check whether they match;
if so, remove both and test shorter string

If last character is not a letter remove it and test shorter string

If first character is not a letter remove it and test shorter string

Implementation of a `isPalindrome` method (V)

Find solutions to the simplest inputs.

Strings with two characters No special case required; step two still applies

Strings with a single character They are palindromes

The empty string It is a palindrome

Implementation of a isPalindrome method (VI)

Implement the solution by combining the simple cases and the reduction step

```
public static boolean isPalindrome(String text){
   int length = text.length();
   // Separate case for shortest strings.
   if (length <= 1) { return true; }
   else {
      // Get first and last characters, converted to lowercase.
      char first = Character.toLowerCase(text.charAt(0));
      char last = Character.toLowerCase(text.charAt(length - 1));
      if (Character.isLetter(first) && Character.isLetter(last) {
         // Both are letters.
         if (first == last) {
            // Remove both first and last character.
            String shorter = text.substring(1, length - 1);
            return isPalindrome(shorter);
         } else
            return false;
      } else if (!Character.isLetter(last)) {
         // Remove last character.
         String shorter = text.substring(0, length - 1);
         return isPalindrome(shorter);
      } else {
         // Remove first character.
         String shorter = text.substring(1);
         return isPalindrome(shorter);
      }
   }
}
```

Helper methods

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem
- Consider the palindrome test of previous section
- It is a bit inefficient to construct new string objects in every step

Substring Palindromes I

Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:

```
/**
   Tests whether a substring is a palindrome.
   @param text a string that is being checked
   @param start the index of the first character of the substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
public static boolean isPalindrome(String text, int start, int end)
```

Substring Palindromes II

Then, simply call the helper method with positions that test the entire string:

```
public static boolean isPalindrome(String text) {
   return isPalindrome(text, 0, text.length() - 1);
}
```

Substring Palindromes III

```java
public static boolean isPalindrome(String text, int start, int end) {
   // Separate case for substrings of length 0 and 1.
   if (start >= end) { return true; }
   else {
      // Get first and last characters, converted to lowercase.
      char first = Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end));
      if (Character.isLetter(first) && Character.isLetter(last)) {
         if (first == last)
            return isPalindrome(text, start + 1, end - 1);
         else return false;
      } else if (!Character.isLetter(last)) {
         return isPalindrome(text, start, end - 1);
      } else {
         return isPalindrome(text, start + 1, end);
      }
   }
}
```

We will discuss further the topic of recursion once we have covered more on classes and types. . .

Summary I

- A method is a named sequence of instructions.
- Actual parameters are supplied when a method is called.
- The return value is the result that the method computes.
- When declaring a method, you provide a name for the method, a variable for each formal parameter, and a type for the result.
- Method comments explain the purpose of the method, the meaning of the parameters and return value, as well as any special requirements.
- Variables hold the arguments supplied in the method call.

Summary II

- The `return` statement terminates a method call and yields the method result.
    - Turn computations that can be reused into methods.
    - Use a return type of `void` to indicate that a method does not return a value.
- Use the process of *stepwise refinement* to decompose complex tasks into simpler ones.
    - When you discover that you need a method, write a description of the parameter variables and return values.
    - A method may require simpler methods to carry out its work.

Summary III

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

- Two local or parameter variables can have the same name, provided that their scopes do not overlap.
- You can use the same variable name within different methods since their scope does not overlap.
- Local variables declared inside one method are not visible to code inside other methods

Summary IV

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.
- The key to finding a recursive solution is reducing the input to a simpler input for the same problem.
- When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

Questions