Mat 2170
Week 7

Methods – Algorithms

Spring 2014
Student Responsibilities

Reading: Textbook, Sections 5.2 – 5.5

Lab

Attendance

Overview Chapter Five, Sections 2 — 5:

- 5.2 Writing your own methods
- 5.3 Mechanics of the method–calling process
- 5.4 Decomposition
- 5.5 Algorithmic methods
5.2 Writing Our Own Methods

The general form of a method definition is:

```
scope  type  name (argument list)
{
    statements in the method body
}
```

where

- **scope**: indicates what blocks of code have access to the method choices: **public**, **private**, or **protected**
- **type**: indicates the type of value the method returns (if any)
- **name**: is the name of the method
- **argument list**: is the ordered list of declarations for the variables used to hold the values of each argument
Scope and Type

```
scope type name (argument list) {
    statements in the method body
}
```

- **Scope**: what code blocks have access?
  1. The most common value for `scope` is **private**, which means that the method is available only within its own class.
  2. If other classes need access to the method, `scope` should be **public** instead.

- **Type** should be **void** if a method does **not** return a value. Such methods are sometimes called **procedures**.

- If a method has a return type other than **void**, then it **must** return a value.
Returning Values from a Method

You can return a **single** value from a method by including a `return` statement, which is usually written as:

```
return expression;
```

where `expression` is a Java expression that specifies the value the method is to return.

As an example, the method definition:

```
private double feetToInches (double feet) {
    return 12.0 * feet;
}
```

converts an argument indicating a distance in feet to the equivalent number of inches, and returns this calculated value to the calling program.
Methods Involving Control Statements

- The **body** of a method can contain statements of any type, including control statements: `for`, `while`, `if`, and `switch`.

- As an example, the following method uses an `if` statement to find the larger of the two integer arguments:

```java
private int MyMax (int x, int y) {
    if (x > y)
    {
        return x;
    }
    else // x <= y
    {
        return y;
    }
}
```

- **Return** statements can be used **at any point** in the method, and may **appear more than once**, although **only one** will be executed during a particular call.
The factorial Method

- The **factorial** of a number \( n \) (written as \( n! \)) is defined to be the product of the integers from 1 to \( n \). Thus, \( 5! \) is \( 1 \times 2 \times 3 \times 4 \times 5 \), or 120.

- The following method definition uses a **for** loop to compute the factorial function:

```java
private int factorial (int n) {
    int result = 1;
    for (int i = 2; i <= n; i++)
    {
        result *= i;
    }
    return result;
}
```

- Note here that the accumulator **result** stores a **product** rather than a sum, so it must be initialized to 1 instead of 0.
Non–numeric Methods

Methods in Java can return values of any type. The following method, for example, returns the English name of the day of the week, given a number between 0(Sunday) and 6(Saturday):

```
private String weekdayName (int day) {
    switch (day) {
    case 0:  return "Sunday";
    case 1:  return "Monday";
    case 2:  return "Tuesday";
    case 3:  return "Wednesday";
    case 4:  return "Thursday";
    case 5:  return "Friday";
    case 6:  return "Saturday";
    default: return "Illegal weekday";
    }
}
```

(String is a class defined in the package java.lang.)

There is no need for a break statement following a return.
Methods Returning Graphical Objects

- Textbook has examples of these types of methods.

- The following method creates a filled circle centered at the point \((x, y)\), with a radius of \(r\) pixels, and is filled using the color specified in the parameter list.

```java
private GOval createFilledCircle (double x, double y, double r, Color color) {
    GOval circle = new GOval(x-r, y-r, 2*r, 2*r);
    circle.setFilled(true);
    circle.setColor(color);
    return circle;
}
```

- If you are creating a GraphicsProgram that requires many filled circles in different colors, the createFilledCircle() method turns out to save a considerable amount of code.
**Predicate Methods**

- Methods that return a `boolean` value play an important role in programming and are called **predicate methods**.

- As an example, the following method returns `true` if the first argument is divisible by the second, and `false` otherwise:

```java
private boolean isDivisibleBy (int x, int y) {
    return x % y == 0;
}
```

Notice that when `x` is evenly divisible by `y`, `true` is returned, otherwise `false` is returned — an if statement isn’t required in this case.
Once you have defined a predicate method, you can use it just like any other Boolean value.

For example, you can print the integers between low and high that are divisible by 7 by running a for loop through the integers [low..high] and checking which are divisible by 7:

```java
for (int i = low; i <= high; i++)
{
    if (isDivisibleBy(i, 7))
    {
        println(i);
    }
}
```

Notice that numbers which aren’t divisible by 7 are simply ignored.
Using Predicate Methods Effectively

- While the following code is not incorrect, it is **inelegant**:

```java
private boolean isDivisibleBy (int x, int y) {
    if (x % y == 0) {
        return true;
    } else {
        return false;
    }
}
```

- A similar problem occurs when beginning programmers include an explicit comparison in an `if` statement to see if a predicate method returns `true`.

**Avoid redundant tests** such as this:

```java
if (isDivisibleBy(i, 7) == true)
```
Method: Powers of Two

- The following method takes an integer \( n \) and returns \textbf{true} if \( n \) is a power of two, and \textbf{false} otherwise.

- The powers of 2 are: 1, 2, 4, 8, 16, 32, and so forth; numbers that are less than or equal to zero cannot be powers of two.

```java
private boolean isPowerOfTwo (int n) {
    if (n < 1) return false;
    while (n > 1) {
        if (n % 2 == 1) return false;
        n /= 2;
    }
    return true;
}
```

- If \textbf{at any time} it is discovered that the value is \textbf{not} a power of 2, \textbf{false} is returned. If execution drops out of the loop, then the original number was a power of 2, and \textbf{true} is returned.
5.3 Mechanics of the Method–Calling Process

When you invoke a method the following actions occur:

- The argument expressions are evaluated (in the context of the calling method)

- Each argument value is copied into the corresponding parameter variable, which is allocated in a newly assigned region of memory called a stack frame.

This assignment follows the order in which the arguments appear: the first argument is copied into the first parameter variable, and so on.
Mechanics of the Method–Calling Process, Cont.

- The statements in the method body are evaluated (using the new stack frame to look up the values of local variables).

- When a return statement is encountered, it computes the return value and substitutes that value in place of the original call.

- The stack frame for the called method is discarded, and execution is returned to the calling program, continuing from where it left off.
The Combinations Function

- To illustrate method calls, the text uses a function $C(n, k)$ that computes the combinations function — the number of ways one can select $k$ elements from a set of $n$ objects.

- Suppose, for example, that you have a set of five coins:

- How many ways are there to select two coins?
  - penny + nickel
  - nickel + dime
  - dime + quarter
  - quarter + dollar
  - penny + dime
  - nickel + quarter
  - dime + dollar
  - penny + quarter
  - nickel + dollar
  - penny + dollar

  for a total of 10 ways.
Combinations and Factorials

- Fortunately, mathematics provides an easier way to compute the combinations function than by counting out all the ways.

- The value of the combinations function is given by the formula:

\[ C(n, k) = \frac{n!}{k! \times (n - k)!} \]

- Given that we already have a `factorial()` method, it is easy to turn this formula directly into a Java method:

```java
private int combinations (int n, int k)
{
    return factorial(n) / (factorial(k) * factorial(n-k));
}
```
The Combinations Program

```java
public void run()
{
    println("Program to calculate combinations");
    int num = readInt("Enter number objects in set: ");
    int chosen = readInt("Number to be chosen: ");
    println("C(" + num + ", " + chosen + ") = " +
             combinations(num,chosen));
}
```

![Applet Viewer: TestCombination.class](image-url)
5.4 Decomposition

One of the most important advantages of methods is that they make it possible to break a large task down into successively simpler pieces. This process is called decomposition.

- Once you have completed the decomposition, you can then write a method to implement each subtask.
Choosing a Decomposition Strategy

- One of the most subtle aspects of programming is the process of **deciding how to decompose** large tasks into smaller ones.

- In most cases, the best decomposition strategy for a program follows the structure of the real–world problem that program is intended to solve.

- If the problem seems to have natural **subdivisions**, those subdivisions usually provide a useful basis for designing the program decomposition.

- Each subtask in the decomposition should **perform a function** that is **easy to name and describe**.
Decomposition Goals

- One of the primary goals of decomposition is to **simplify the programming process**.

- A good decomposition strategy must **limit the spread of complexity**.

- Each level in the decomposition should **take responsibility** for **certain details**, and avoid having those details percolate up to higher levels.

For example, in the program to calculate the combinations, the problem was broken down to utilize the `factorial()` method. Thus, the `combinations()` method was less cluttered and easier to read.
5.5 Algorithmic Methods

- **Methods** are important in programming because they provide a structure in which to express algorithms.

- **Algorithms** are abstract expressions of a solution strategy.

- **Implementing** an algorithm as a method makes that abstract strategy concrete.

- Algorithms for solving a particular problem can vary widely in their efficiency — it makes sense to think carefully when choosing an algorithm because making a bad choice can be extremely costly.
Greatest Common Divisor

- Section 5.5 in the text looks at two algorithms for computing the greatest common divisor of two integers.

- The GCD is defined to be the largest integer that divides evenly into both

- There is big difference in the efficiency of the two algorithms: brute force vs Euclid’s algorithm.
Brute–Force Approach

■ **Trying every possible solution** is called a **brute–force** strategy.

■ For the greatest common divisor, we can count backwards from the smaller of the two numbers until we find a value that divides both numbers evenly.

```java
public int gcd(int x, int y) {
    int guess = Math.min(x, y);
    while (x % guess != 0 || y % guess != 0) {
        guess--;
    }
    return guess;
}
```
This gcd() algorithm **must terminate** for positive values of \( x \) and \( y \) because the value of **guess** will eventually reach 1 if it doesn’t stop before that.

At the point it terminates, **guess** must be the greatest common divisor because the while loop will have already tested all larger possibilities and discarded them.

Note that in the worst case, when the \( \text{gcd}(x, y) \) is 1, the loop must iterate all the way from the smaller of the two numbers down to 1.

Computing \( \text{gcd}(1000005, 1000000) \) results in **almost a million** steps to obtain the answer, 5.
Euclid’s Algorithm

- A better, more **efficient** algorithm can produce an answer more quickly.

- The mathematician Euclid of Alexandria described a more efficient algorithm 23 centuries ago:

```java
public int gcd(int x, int y) {
    int r = x % y;
    while (r != 0) {
        x = y;
        y = r;
        r = x % y;
    }
    return y;
}
```

- Using Euclid’s algorithm, the gcd(1000005, 1000000) takes **two** steps.
How Euclid’s Algorithm Works

- Euclid’s great insight was that the greatest common divisor of \( x \) and \( y \) must also be the greatest common divisor of \( y \) and the remainder when \( x \) is divided by \( y \).

- He was able to prove this proposition in Book VII of his *Elements*

- The next slide works through the steps geometrically to illustrate the calculation when \( x \) is 78 and \( y \) is 33.
An Illustration of Euclid’s Algorithm

Step 1: Compute the remainder of 78 divided by 33:

\[ x \begin{array}{c} \text{78} \\ \end{array} \]
\[ y \begin{array}{c} \text{33} \end{array} \]
\[ y \begin{array}{c} \text{33} \end{array} \]
\[ y \begin{array}{c} \text{12} \end{array} \]

Step 2: Compute the remainder of 33 divided by 12:

\[ x \begin{array}{c} \text{33} \\ \end{array} \]
\[ y \begin{array}{c} \text{12} \end{array} \]
\[ y \begin{array}{c} \text{12} \end{array} \]
\[ y \begin{array}{c} \text{9} \end{array} \]

Step 3: Compute the remainder of 12 divided by 9:

\[ x \begin{array}{c} \text{12} \\ \end{array} \]
\[ y \begin{array}{c} \text{9} \end{array} \]
\[ y \begin{array}{c} \text{3} \end{array} \]

Step 4: Compute the remainder of 9 divided by 3:

\[ x \begin{array}{c} \text{9} \\ \end{array} \]
\[ y \begin{array}{c} \text{3} \end{array} \]
\[ y \begin{array}{c} \text{3} \end{array} \]
\[ y \begin{array}{c} \text{3} \end{array} \]

Because there is no remainder, the answer is 3.
In graphical programs there are two strategies for providing methods with size and location information:

1. Use shared **named constants** to define the picture parameters
2. Pass the information as **arguments** to each method

Using named constants is easy, but relatively inflexible. If you define constants to specify the location of an object, you can only draw the object at that location.
Using arguments is more cumbersome, but makes it easier to change such values.

It is best to find an appropriate trade-off between the two approaches. The text recommends:

- Use **arguments** when callers need to supply different values
- Use **named constants** when there is a known satisfactory value