

Chapter 2: Basic Programming

Up to now we created programs that defined the standard program entry point `public static void main` and executed their code from that point on. We learned how to manipulate the flow of execution using `if`, `for`, and `while` constructs and how to use the basic data types in Java. While that approach works fine for small projects, it is not sufficient for medium or large-scale projects.

Programs to solve more complicated tasks group segments of code into functional units called methods and use custom-made types that include data elements and operations to perform work. This chapter will focus on creating complete programs using methods while chapter 3 will show how to create new types called classes.

Section 2.1 introduces methods and shows how data can flow into and out of them. Section 2.2 focuses on the life span of variables called their scope. Section 2.3 presents the new data type array to group variables of the same type together into blocks. Section 2.4 focuses on creating 'good' programs by providing suggestions for using proper comments and for testing your code. It includes a `Console` class to easily obtain user input. In section 2.5 we learn to embed simple data types in wrappers to enhance their functionality. The last section, which is optional, provides a case study that pulls together most of the concepts introduced so far to create a program that solves a system of linear equations.

Quick View

Here is a quick overview of the topics covered in this chapter.

2.1. Methods and Parameters

Methods; Parameter Passing; Reference Types as Parameters

2.2. Fields and Scope

Fields and Local Variables; Scope Rules

2.3. Arrays

Declaring and Initializing Arrays; Arrays as Input Parameters; Higher-Dimensional Arrays

2.4. Documenting, Testing, and Keyboard Input

Implementation and Documentation Comments; The `Console` Class for Keyboard Input; Testing and Debugging Code

(*) 2.5. Wrapper Classes for Basic Data Types**(**) Case Study: Solving a System of Linear Equations**

(*) This section is optional but recommended

(**) This section is optional

2.1. Methods and Parameters

Java provides a variety of mechanisms to break up a project into smaller units, each responsible for handling a well-defined part of the overall goal. The best and most versatile approach uses *classes* to define these units, but for now we will focus on a more elementary approach.¹

Methods

Methods

A method² is a named segment of code that performs a small, well-defined task. It has a header that specifies how to exchange information with other methods and a body that defines what it is doing. Methods can call other methods, or in turn be called by other methods. Methods can have input parameters defined in the header to adjust their behavior and they can return a single value to the entity calling them.

Methods are defined inside the body of a class. They have the following syntax:

```
[ public static ] returnType methodName(parameterList)
{ /* method body */ }
```

where

- *public static* are keywords that at this stage must be present³
- *returnType* is a known data type or the special keyword `void` for no return type
- *methodName* is a valid Java name
- *parameterList* is a comma-separated list of typed input parameters, or empty

We have already used a method named `main` in every complete program of chapter 1:

```
public class ClassName
{ public static void main(String args[])
  { /*... more code ... */ }
}
```

To phrase this basic framework in a different way, we could say:

Java Program

A Java program is a named class that contains at least the keywords `public class ClassName` and at least one method inside the class body called `main` with the method header `public static void main(String args[])`. Execution of the program begins with that method.

```
public class ClassName
```

¹ We will define *class* in chapter 3.

² Other programming languages use the term *function* or *procedure* instead of method. Pascal, for example, calls methods without a return value *procedures*, those with return values *functions*.

³ In chapter 3 we will explain what the significance of `static` is and why it must be present at this stage.

```

{ /* other methods as needed */
  public static void main(String args[])
  { /* the body of the main method */ }
}

```

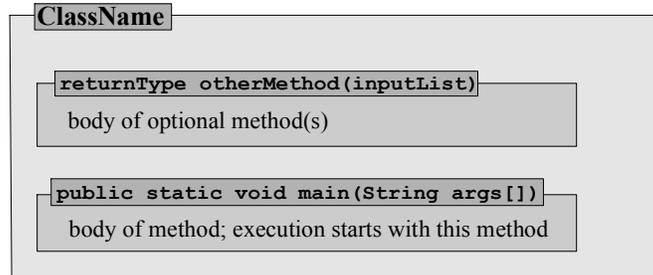


Figure 2.01: Representation of a Java Program

The beginning of a method is at its header, but execution of a method can stop at various points inside the method body.

Method Exit Point

A method has exactly one entry point defined by the method header but it can have multiple exit points where the method stops executing and returns control to another entity. An exit point can be defined using the keyword `return` and methods returning `void` have a default exit point:

- *If the `returnType` of the method is not `void`, the keyword `return` must occur at least once in the method body and must return a value of the type specified as `returnType` in the method header. Once a `return` statement executes, no other code of the method executes.*
- *If the `returnType` is `void` and the keyword `return` is not present, then the last line in the block of the method is the default exit point. You can create alternate exit points by using the keyword `return` by itself.*

Example 2.01: Using methods to compute properties of a square

Create a complete program to compute the area and perimeter of a square. Use a separate method for each computation.

Problem Analysis: We can divide this task into three subtasks:

1. Compute the area of a square, given its side length
2. Compute the perimeter of a square, given its side length
3. Decide when and how to apply (1) and (2) and display the results

Step 1: The method to compute the area requires one input parameter for the side length of the square and returns its area:

```

public static double computeArea(double side)
{ return side * side; }

```

In other words, we define a method named `computeArea` that takes as input a double value named `side` (representing the side length of a square) and produces as its result the value of `side * side`.

Step 2: The second method is closely related. It uses one input parameter and returns the perimeter of the square:

```
public static double computePerimeter(double side)
{ return 4.0 * side; }
```

Step 3: The main method puts everything together and decides when to invoke the two methods and what to do with the values they return:

```
public static void main(String args[])
{ double side = 10.0;
  double area = computeArea(side);
  double perim = computePerimeter(side);
  System.out.println("Side: " + side);
  System.out.println("Area: " + area);
  System.out.println("Perimeter: " + perim);
}
```

We can combine the three methods into a working program:

```
public class SquareProperties
{ public static double computeArea (double side)
  { return side * side; }
  public static double computePerimeter(double side)
  { return 4.0 * side; }
  public static void main(String args[])
  { double side = 10.0;
    double area = computeArea(side);
    double perim = computePerimeter(side);
    System.out.println("Side: " + side);
    System.out.println("Area: " + area);
    System.out.println("Perimeter: " + perim);
  }
}
```

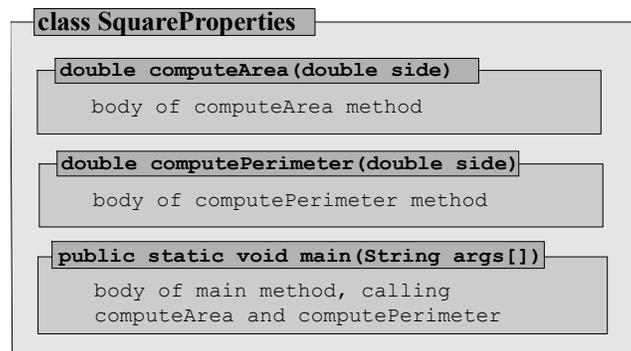


Figure 2.02: A representation of the SquareProperties class

At first it might appear as overkill to use three methods to solve this simple problem. We could handle all the work in a single main method and the code would be shorter. But classes that only contain one method are usually ill designed and not as flexible as they should be. In our example we can reuse the methods `computeArea` and `computePerimeter` every time we need to compute the area and perimeter of a square, even in different classes or programs.

Software Engineering Tip: Methods are particularly well suited to solve small, well-defined subsets of a bigger task. We can and should subdivide a task we want to solve, break it up into little subtasks, and then solve each subtask using individual methods. This process is called modularization. As a rule, each method should be as simple and as short as possible.

Breaking up larger tasks into subtasks works especially well for more difficult problems:

Example 2.02: Using methods to find the sum of prime numbers

In example 1.33, we created a program to find the sum of all prime numbers less than 10,000. Rewrite this program, using one or more appropriate methods.

Problem Analysis: There are clearly two subtasks to perform here:

1. find out whether a given number is prime
2. add prime numbers less than 10000 to a running total

Step 1: The first subtask calls for a method returning `true` if the input number is prime and `false` if it is not. We already saw the basic code for that in example 1.32, so we can simply wrap that code in a method with an appropriate name, input parameter, and return type:

```
public static boolean isPrime(int number)
{
    int i = 2;
    while ((i < number) && ((number % i) != 0))
        i++;
    if (i == number)
        return true;
    else
        return false;
}
```

Step 2: The `isPrime` method does the hard work of testing an input number for being prime and it is now easy to sum up prime numbers less than 10,000:

```
public static void main(String args[])
{
    int primeSum = 1;
    for (int x = 1; x < 10000; x++)
    {
        if (isPrime(x))
            primeSum += x;
    }
    System.out.println(primeSum);
}
```

In the above example the `main` method delivers the answer 5,736,397, which may or may not be correct. If we replace 10,000 by 10 in the `for` loop of the above `main` method and recompile and execute the new program, it delivers the answer 18, which is correct: $1 + 2 + 3 + 5 + 7 = 18$. Having checked our methods in this simple case we are inclined to believe that the original answer 5,736,397 is correct as well.⁴

⁴ On the other hand, if we replace 10,000 by 1,000,000 in our `main` method the program delivers a negative answer. That is obviously incorrect and is caused by integer overflow as explained in chapter 1.

Software Engineering Tip: It is often difficult to determine whether a program works correctly. Therefore we should always test any program in one or more simple cases, verifying the answers by hand. We should also analyze every answer by looking for obvious mistakes.

While the `main` method seems to compute the correct answer, the method `isPrime` is not entirely correct: if the input value is 1 the method will return `false` even though 1 *is* a prime number. In our use of the method it does not make a difference, because we have already taken 1 into account by initializing the variable `primeSum` to 1 before using the `isPrime` method. Since we may at a later time use `isPrime` again we should make sure it works exactly as its name applies. Therefore, we should rewrite it as follows:

```
public static boolean isPrime(int number)
{
    if (number == 1)
        return true;
    else
    {
        int i = 2;
        while ( (i < number) && ((number % i) != 0))
            i++;
        if (i == number)
            return true;
        else
            return false;
    }
}
```

By initializing the variable `primeSum` in the `main` method to zero we are certain that we do not add 1 (which will now be certified as a prime number by `isPrime`) twice. ■

Software Engineering Tip: Choose every method name carefully. It should clearly indicate what the method is doing. If there are even slight differences between the implied meaning of a method name and its actual performance, change the method so that it performs as its name implies.

Parameter Passing

In previous examples we defined and used simple methods that required one input parameter and worked much like standard mathematical functions. Since methods can require more than one input parameter, or none, we need to clarify the process of passing parameters into a method. In particular, what happens if a method receives an input value and then changes that value? Will that change be visible outside the method?

Input Parameters for Methods

When calling a method with an empty input parameter list, use the method name and a set of empty parentheses.

When calling a method with a non-empty input parameter list, use the method name and as many comma-separated expressions as there are parameters in the input list. Each expression must be of the same or compatible type as the corresponding input parameter.

All input parameters for methods are value parameters where the value of the input expression is copied into the corresponding variable in the method header. Any change that the method causes to its variables is not visible outside that method.⁵

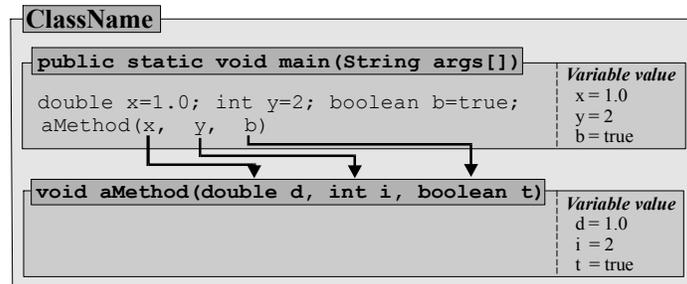


Figure 2.03: Matching input parameters and variables to call method

Software Engineering Tip: Some programming languages offer a choice to declare input variables as value parameters (where the value is copied into a new variable) or as reference parameters (where two variables refer to the same data). Java offers only value parameters and changes that should find their way out of the method must be specified using `return`.⁶

Example 2.03: Illustrating value parameters

Create a program that contains one method with one input parameter and `void` return type in addition to the `main` method. Change the value of the input parameter in the method and check whether this change is visible in the `main` method.

Let's define a method with an integer input parameter, increment the value of that variable inside the method, then see if that change is visible to the `main` method by displaying the value of the original variable.

```
public class ValueParameter
{
    public static void aMethod(int input)
    {
        System.out.println("Value of input at start of aMethod: " + input);
        input += 10;
        System.out.println("Value of input at end of aMethod: " + input);
    }
    public static void main(String args[])
    {
        int number = 10;
        System.out.println("Value before calling method: " + number);
        aMethod(number);
        System.out.println("Value after calling the method: " + number);
    }
}
```

Figure 2.04 shows the result after running this program:

⁵ A method *can* change the *state* of an object that a parameter refers to, which *is* visible outside the method.

⁶ See “Reference Types as Input Parameters” for details.

```

Execution Results *
C:\>java ValueParameter
Value before calling method: 10
Value of input at start of aMethod: 10
Value of input at end of aMethod: 20
Value after calling the method: 10

```

Figure 2.04: Execution Results for ValueParameter.java

The variable `number` is initialized to 10 in the main method, then this value is copied into the variable `input` in the method header of `aMethod`.

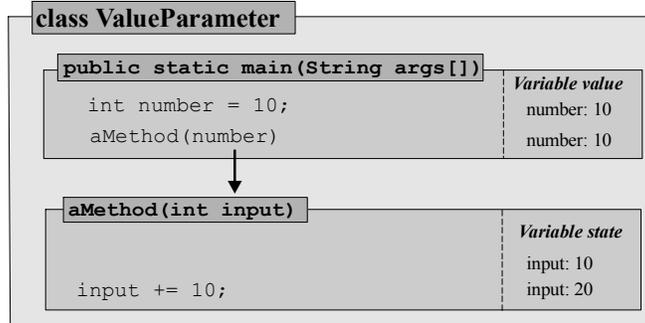


Figure 2.05: Parameter passing

The method changes the value of `input` to 20, but upon exiting the value of `number` remains 10. The value of `number` is copied into `input` but they are separate entities and can change independently. ■

Results of a computation or a method can also be used as input parameters as long as they are type-compatible.

Example 2.04: Illustrating valid input expressions

Look at the program listed below. If the compiler would flag an error, please explain that error. Discard lines that result in a compiler error and list the output of the program.

```

1    public class Squared
2    {    public static double square(double x)
3        {    return x*x; }
4    public static int cube(int i)
5        {    return i*i*i;}
6    public static double power(double base, int exp)
7        {    double result = 1;
8            for (int i = 1; i <= exp; i++)
9                result *= base;
10           return result;
11        }
12    public static void main(String args[])
13    {    double x = 2.0;
14        double y = square(square(x));
15        System.out.println("x: " + x + ", y: " + y);
16
17        int i = 2;
18        int j = cube(cube(cube(i)));
19        System.out.println("i: " + i + ", j: " + j);
20
21        int z = cube(square(i));

```


Reference Types as Parameters

Parameter passing has different side effects if the *state* of an object that a variable points to is modified in a method instead of the variable's value. To understand this we need to explain the difference between reference variables and variables of basic type:

Reference Type

A variable of reference type is a variable whose identifier acts as an access point to the data but does not represent the data itself. A reference variable is declared as usual, but must be initialized using the special keyword `new`.⁷

All types except for the basic data types `short`, `int`, `long`, `float`, `double`, `char`, and `boolean` are reference types.

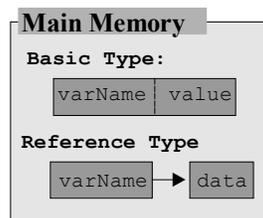


Figure 2.06: Reference Type

The only reference types we currently know are `String` and `StringBuffer`. Strings are an exception to the rule that reference variables must be initialized using `new`, but they *are* reference types.

Software Engineering Tip: It is important to understand that a reference variable only acts as access point to data and does not represent the data directly⁸:

- If you change the value of a reference variable it will refer to data located at a different memory location.
- If you change the data pointed to by the reference variable, the variable itself does not change, but the data does.

One of the manifestations of the difference between reference types and simple types is that variables of reference type can not be checked for equality using the double-equal operator.

Checking Reference Variables for Equality

When using the double-equal `==` or not equal `!=` operators to check two reference variables for equality, the operators check whether both variables refer to the same object in memory, not

⁷ In other programming languages this type of reference variable is called a pointer. Java is sometimes said not to use pointers, but the contrary is true: *All* types except for basic types are pointers in Java, but you can use them as if they were regular variables. One of Java's strengths is that it disguises the use of pointers so effectively.

⁸ In many programming languages pointers require special syntax construction to access the data they refer to. Java uses the dot operator for that purpose.

whether the objects themselves are equal. Only when two reference variables refer to the same object in memory does the double-equal operator evaluate to true.

This explains the strange behavior we encountered in example 1.35 where we compared two strings that should have been the same with the double-equal operator. Here is a similar example:

Example 2.05: Equality of String types

Suppose four variables are defined as in the code below:

```
String s1 = "Java by Definition";
String s2 = new String("Java by Definition");
String s3 = s2;
String s4 = "Java " + new String("by Definition");
```

Which ones would check out as equal using the double-equal operator?

These strings are equal in the sense that they contain the same characters in the same order. The string operation "equals" as described in section 1.4 would show that all of them are equal in that sense. However, that's not what the double-equal operator does: it checks instead which of these reference variables refer to the same object in memory. Figure 2.07 shows the data to which our reference variables point:

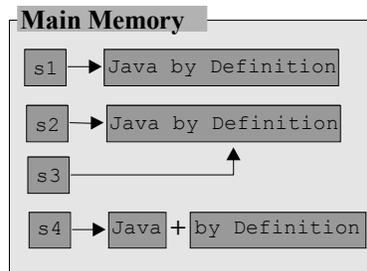


Figure 2.07: Strings as Reference Types

Since `s3` is assigned the value of `s2`, both refer to the same data in memory and `(s2 == s3)` should return `true`. To see which other variables are "double-equal" we use a small program whose output is shown in figure 2.08.

```
public class StringReferences
{ public static void main(String args[])
  { String s1 = "Java by Definition";
    String s2 = new String("Java by Definition");
    String s3 = s2;
    String s4 = "Java " + new String("by Definition");
    System.out.println("s1 == s2: " + (s1 == s2));
    System.out.println("s1 == s3: " + (s1 == s3));
    System.out.println("s1 == s4: " + (s1 == s4));
    System.out.println("s2 == s3: " + (s2 == s3));
    System.out.println("s2 == s4: " + (s2 == s4));
    System.out.println("s3 == s4: " + (s3 == s4));
  }
}
```

```

C:\>java StringReferences
s1 == s2: false
s1 == s3: false
s1 == s4: false
s2 == s3: true
s2 == s4: false
s3 == s4: false

```

Figure 2.08: Execution Results for StringReferences

Only `s2` and `s3` point to the same object in memory, all other strings are located at different memory locations.

Reference Types as Input Parameters

When a variable of reference type is used as input parameters to a method, its value, as usual, is copied. If the method changes the value of the reference variable, that change is not visible to the calling method. But if the method modifies the state of the data that the input variable refers to, that change is visible to the calling method.

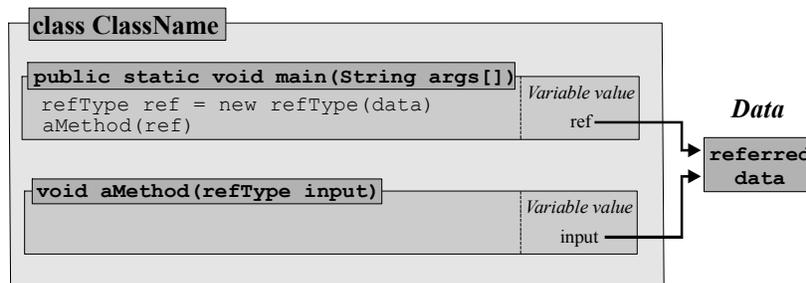


Figure 2.09: Reference type as method parameter

To provide an example we need to use either `String` or `StringBuffer`, the only reference types we currently know.

Example 2.06: Using reference types as method input parameters

Create a program with two methods. The standard `main` method initializes two variables of type `String` and `StringBuffer`, respectively, which are then used as input to the second method. That method appends a string to both input variables. Which changes are visible to the `main` method?

We need one `main` method to declare and initialize the variables and to call a second method with those variables as input. The second method appends a value to both variables, using the `append` method for the `StringBuffer` and the concatenation operator `+` for the `String` type, as explained in section 1.4:

```

public class StringTester
{
    public static void changeStrings(String s, StringBuffer sb)
    {
        s += " by Definition";
        sb.append(" by Definition");
    }
    public static void main(String args[])
    {
        String string = new String("Java");
        StringBuffer buffer = new StringBuffer("Java");

```

```

changeStrings(string, buffer);
System.out.println("String after method call: " + string);
System.out.println("StringBuffer after method call: " + buffer);
}
}

```

Figure 2.10 explains the situation when the program executes:

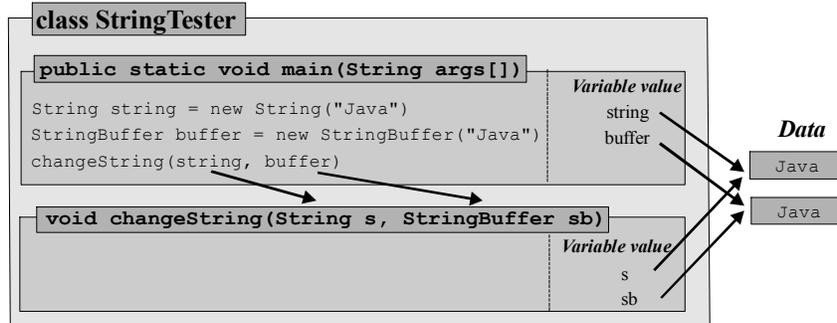


Figure 2.10 (a): `StringTester` before `changeString` executes

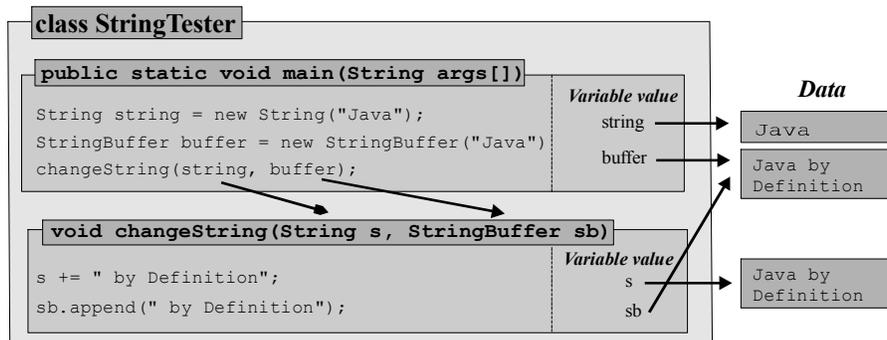


Figure 2.10 (b): `StringTester` after `changeString` executes

As figures 2.10 (a) and (b) illustrate, the output of the program is:

```

C:\>java StringTester
String after method call: Java
StringBuffer after method call: Java by Definition

```

Inside `changeString` the `String` " by Definition" is concatenated to `s`, resulting in a *new* `String` with value "Java by Definition". The variable `s` is then reset to point to that new data, but that change is *not* visible outside the method. The variables `s` and `string` now refer to different data values. The variable `sb`, on the other hand, uses the method `append` to modify the state of the data it refers to. The value " by Definition" is added to that data while the value of `sb` does not change. Since `sb` and `buffer` still refer to the same data, the change to the data *is* visible outside the method. ■

Software Engineering Tip: It is easy to see when the value of a reference variable changes and when the data it refers to changes:

- If a reference variable is on the *left* side of an assignment operator, its value changes. Such changes *are not* visible outside.

- If a reference variable is used with the dot operator, only the data can change. Such changes *are* visible outside.

2.2. Fields and Scope

We have so far declared variables as needed, anywhere in our program and in method headers, but we did not mention how long these variables remain valid and accessible. As it turns out, Java has two types of variables as far as their longevity is concerned.

Fields and Local Variables

Variables can be defined inside a method, as we have done many times, as well as outside a method:

Fields and Local Variables

In Java, there are two types of variables:

- *Variables that are defined inside a class but not inside a method. These variables are called fields. The type of a field, currently, needs to be prefaced with the keyword `static`⁹. All methods of a class have direct access to field variables.*
- *Variables that are declared inside a method when needed. These variables are called local variables. Input variables declared in the method header are local variables of that method.*

Figure 2.11 shows a class with one field named `aField` and four local variables named `args`, `localX`, `localY`, and `localZ`. All variables are properly initialized except for `args`. The initial value of `localY` is 1.0, since that value is copied from `localX` when `aMethod` is called.

```
class ClassName
{
    double aField = 0.0;
    public static void main(String args[])
    {
        double localX = 1.0;
        aMethod(localX);
    }
    void aMethod(double localY)
    {
        double localZ = 3.0;
    }
}
```

Figure 2.11: A class with one field and four local variables

Software Engineering Tip: Traditionally, fields are defined either at the beginning of the class before defining methods, or at the end of the class after all methods have been defined. We

⁹ Later, that restriction will no longer apply. As we will see in chapter 3, `static` methods can only refer to `static` fields and other `static` methods. Since all our programs start from the `static main` method, all fields and methods must be `static` at this stage.

recommend placing all fields at the *beginning* of a class so that they can be identified immediately. Fields, just as local variables, should be initialized as they are declared.

Example 2.07: Illustrating fields and local variables

Define a short class with one field, one local variable, and one `for` loop. How many fields and local variables does your class have?

A variable is defined as a field by placing its definition outside any method and prefacing it with the keyword `static`. Local variables are declared as usual. Here is our code:

```
public class FieldsAndVars
{   static double aField = 10.0;

    public static void main(String args[])
    {   double int aLocalVar = 20;
        for (int i = 0; i < 10; i++)
            System.out.println("i = " + i);
    }
}
```

There is clearly one field (`aField`) and one local variable (`aLocalVar`). But the variable `i` is also a local variable, as well as the input variable `args[]` of the `main` method. ■

We now have four different types of variables:

Basic Types:	variables that store data directly
Reference Types:	variables that point to data in memory
Local Variables:	variables declared inside method
Fields:	variables that are declared outside all methods and accessible to all methods

Scope Rules

The main difference between fields and local variables is their accessibility, which is explained in the scope rules.

Scope Rules

The scope of a variable describes its longevity, or where in the code the variable is valid.

- *Fields can be used anywhere in the class in which they are defined.¹⁰*
- *Local variables are valid in the block in which they are defined.*
- *Local variable names within one block must be unique*
- *Duplicate field names are not allowed*

¹⁰ In chapter 3 we will learn that fields in one class can be referred to in other classes. Java has additional keywords such as `public` or `private` to fine-tune access to fields across classes.

If a variable is declared twice, once as a field and again as a local variable, the closest or most local definition counts and overrides the previous one.

Figure 2.12 shows an example of how these scope rules govern the validity of fields and variables.

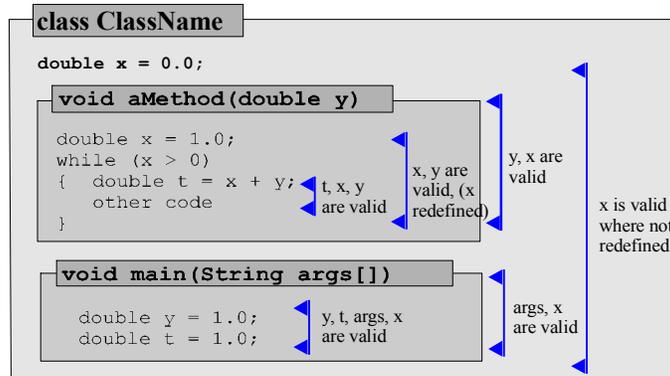


Figure 2.12: Illustrating the scope of variables and fields

Example 2.08: Scope of fields and local variables

Where are the variables (fields and local ones) of example 2.07 valid?

That example contained one field `aField`, which is valid everywhere in the class. If there were more methods aside from `main`, each one could access `aField`.

The local variables `aLocalVar` and `args[]` are declared within the block of the `main` method so they can be accessed everywhere in the `main` method, including inside the `for` loop.

The local variable `i` is defined in the `for` loop. Therefore, it is valid inside that loop only and cannot be accessed outside the loop. ■

Software Engineering Tip: If you need access to the value of a loop counter after a loop is over, you need to declare the variable before the loop starts. A `while` loop works well in that situation because its loop counter is initialized *outside* the loop and has therefore wider scope.

Example 2.09: Compound interest computation using fields

Write a program to compute the interest and balances if \$10,000.00 are deposited for 10 years in an account at an interest rate of 6.5%, compounded once per year. The program should display the interest for each year as well as the yearly amount including interest.

Problem Analysis: We first have to figure out the formulas involved. If you deposit amount dollars at an interest rate of rate (in percent), then the interest after one year is

```
interest = amount * rate / 100
```

and the new amount is the old amount plus that interest

```
amount += interest
```

If we repeat that computation 10 times we have found the compounded interest after 10 years.¹¹ This indicates that we can divide our problem into two subtasks.

1. We need to compute the interest for one year and add it to the current amount
2. We need to repeat this computation 10 times (for the 10 years)

Program Implementation: We define two methods, one called `computeInterest` to compute the interest and the new amount, and another method called `main` to repeat this computation 10 times. The first method needs to know the interest rate as well as the current amount and should deliver the computed interest as well as the new amount. That is a problem, since methods can only return *one* value. Therefore we use a field for the yearly interest as well as the current amount. Since fields are accessible to all methods, the current amount does not have to be an input parameter to the method any longer. The second method to repeat the computation 10 times will be our standard `main` method.

To make our program more flexible we use named constants (see section 1.5) as fields for the interest rate and the number of years. Then `computeInterest` does not need the interest rate as input, because fields are available to all methods in a class. In addition, `computeInterest` does not need to return a value because it changes the field `amount`, which is visible to the `main` method.

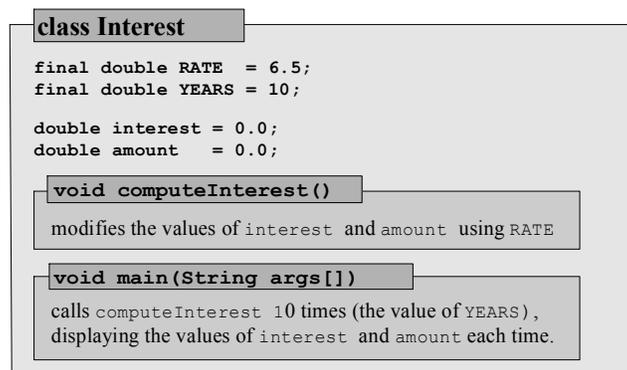


Figure 2.13: Representation of the Interest program

To make sure our output looks nice, we use the decimal formatting tool as described in section 1.5:

```

import java.text.DecimalFormat;

public class Interest
{
    final static double RATE = 6.5;
    final static int YEARS = 10;

    static double interest = 0.0;
    static double amount = 10000.0;

    public static void computeInterest()
    {
        interest = amount * RATE / 100;
        amount += interest;
    }

    public static void main(String args[])
    {
        DecimalFormat formatTool = new DecimalFormat("#,###.00");
    }
}

```

¹¹ The mathematical formula for the compounded interest of x dollars at a rate of $i\%$ after n years, compounded p times per year is $x \cdot \left(1 + \frac{i}{p \cdot 100}\right)^{p \cdot n}$. We could use that formula to verify the answers of our program.

```

for (int i = 1; i <= YEARS; i++)
{ computeInterest();
  System.out.print("Year " + i);
  System.out.print("\tInterest: " + formatTool.format(interest));
  System.out.println("\tTotal: " + formatTool.format(amount));
}
}
}

```

```

C:\>java Interest
Year 1 Interest: 650.00      Total: 10,650.00
Year 2 Interest: 692.25      Total: 11,342.25
Year 3 Interest: 737.25      Total: 12,079.50
Year 4 Interest: 785.17      Total: 12,864.66
Year 5 Interest: 836.20      Total: 13,700.87
Year 6 Interest: 890.56      Total: 14,591.42
Year 7 Interest: 948.44      Total: 15,539.87
Year 8 Interest: 1,010.09    Total: 16,549.96
Year 9 Interest: 1,075.75    Total: 17,625.70
Year 10 Interest: 1,145.67   Total: 18,771.37

```

Figure 2.14: Output of the Interest program

Software Engineering Tip: A natural question at this point is when to use fields and when to use local variables. Since fields are accessible to all methods in a class they can act to transfer information between methods. As a guide¹²:

- Use local variables if they are needed only within a method or are not used to transfer information between methods.
- Use fields if several methods need access to the information stored in variables, or if methods need to return more than one result¹³

Here is a final example illustrating the scope rules when fields and local variables have the same name:

Example 2.10: Illustrating the scope rule

Consider the following program. If a variable is used outside its scope, say so. For all valid variables, state their value at the appropriate `System.out.println` call.

```

1      public class TrickyScope
2      {   static int x = 1;
3
4          public static void method1()
5          {   int x = 10;
6              x += 10;
7          }
8          public static void method2(int y)
9          {   x += 10;
10             y += 20;
11             z += 30;
12          }
13         public static void main(String args[])
14         {   int z = 3;
15             method1();

```

¹² In chapter 3 we will see that fields represent the properties of an object and introduce a better guide to their use.

¹³ Consider using more than one method instead, each returning only one value .

```

16         method2(z);
17         x += 1;
18         y += 2;
19         z += 3;
20         System.out.println(x);
21         System.out.println(y);
22         System.out.println(z);
23     }
24 }

```

Step 1: First we determine which lines are not valid by considering the scope of all variables:

- The variable `x` defined after the class name is a field and valid throughout the class
- The scope for the local variable `x` redefined in `method1` is `method1`
- The scope for variable `y` is `method2`
- The scope for variable `z` (as well as `args`) is the `main` method

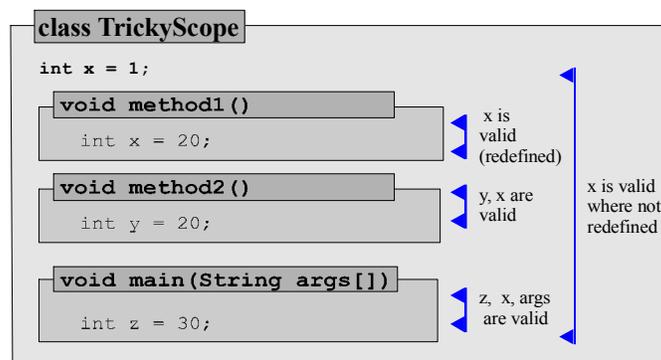


Figure 2.15: TrickyScope.java scope representation

Therefore `method2` uses `z` outside its scope and `main` uses `y` outside its scope so lines 11, 18, and 21 are invalid.

Step 2: Next we determine the values of the valid variables. The first method that executes is `main`. It initializes `z` to 6 and calls `method1`, which declares another local variable named `x`, sets it to 10, then increments it by 10. There are now two variables – one of which is a field – in `method1` called `x` and according to our scope rules only the most local or closest one counts. Therefore, the *field* `x` still has value 1, whereas the *local variable* `x` in `method1` with value 20 expires as soon as the method is done.

Then `method2` executes and the current value of `z` is copied into its local variable `y`. It then adds 10 to the *field* `x`, giving it a value of 11. The local variable `y` is also incremented by 20, making it 23, but that change is invisible to the `main` method.

Finally, the `main` method increments the value of the field `x` again by 1, making it 12. Skipping the invalid statement in line 18, `z` is incremented by 3, making it 6. The final values are:

- `x` has the value 12
- `z` has the value 6

■

2.3. Arrays

So far we have manually declared every variable in a program that is supposed to store information. This makes it difficult to create a program that can store, say, 1000 `double` numbers. We would have to declare variables

```
double num1, num2, ..., num1000;
```

writing out every variable name in between. That, while possible, is of course not practical. Since it is entirely feasible to have a need to store 1000 `double` values (for example, a program to compute the GPA of all Undergraduates of a college), Java offers the possibility to define a whole slew of variables in one declaration.

Declaring and Initializing Arrays

Declaring Arrays

An array is a sequential data structure that can store a fixed number of values of the same type. Every Java array contains an additional variable named `length` to store the current size of the array. Arrays are declared using empty square brackets.

```
type arrayName[];14
```

The default value of an array is `null`. Arrays are reference variables regardless of the types of their elements. They can be used as fields or local variables, or as input or output types of methods.

When you declare an array you do not specify its size but before you can use an array you must define its size to request the appropriate memory from the operating system.

Initializing Arrays

To initialize an array that has previously been declared you use the keyword `new` to define an array of a specific size:

```
arrayName = new type[size];
```

where `size` is an integer expression. You can declare and initialize arrays in one statement:

```
type arrayName[] = new type[size];
```

Individual array elements have their usual default values. You can also implicitly declare and initialize an array by listing its members in a comma-delimited list enclosed in curly brackets:

```
type arrayName[] = { value1, value2, ..., valueN };
```

¹⁴ Java allows an alternative declaration of the form `type[] arrayName`. We will not use that variation.

where value1, value2, ... valueN are of the same type.

Example 2.11: System.out.println applied to arrays

Create an array of 10 doubles, then try to print out the entire array using one System.out.println statement. Does this give the desired result?

We will use the above definitions to declare and initialize an array in one statement, then use System.out.println with the array as argument to see what happens:

```
public class ArrayDisplay
{   public static void main(String args[])
    {   double A[] = new double[10];
        System.out.println(A);
    }
}
```

This code will compile fine but when we run it we get the output shown in figure 2.16:¹⁵



Figure 2.16: Result of using System.out.println on an array

Of course this is not what we expected, but at least it did print out *something*.

Software Engineering Tip: Every data type in Java can be displayed using the System.out.println method. For simple data types the output consists of the values of the variables. For more complicated types, including arrays, the output shows the memory address where the data is stored. That is almost never of interest unless a variable points to null. Using System.out.println can be a convenient way to check the status of variables that may not be initialized correctly during the development stage of a program.¹⁶

Before we can continue, we need to know how to access and manipulate the elements in an array:

Accessing Array Elements

Arrays are indexed with integers starting at zero so that an array of size N has indices 0, 1, 2, ..., N-1. To access the elements of an array A you use square brackets and an integer expression corresponding to the element you want to access:

```
A[intExpression]
```

When you access an array element with an index that is too large or too small it will create a runtime error with the message ArrayIndexOutOfBoundsException.

¹⁵ The output will be different every time you execute the program.

¹⁶ In chapter 3 we will see that we can add a method toString to a class, which will then be used automatically whenever System.out.println is called on a variable of that class type.

Every array also contains a field called `length` that is set to the array's size during initialization. To access that field, you use the array name, then a dot, then the keyword `length`:

```
A.length
```

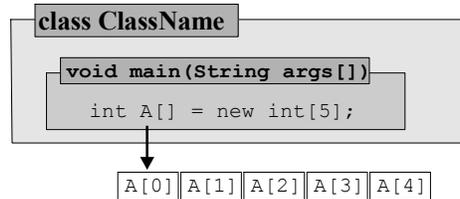


Figure 2.17: Declaring an array of 5 integers

Example 2.12: Displaying an array

In example 2.11 we created an array of 10 doubles and displayed it using `System.out.println`. Since that did not give the desired results, modify the code so that all numbers stored in the array are printed out. Initialize the array elements to the numbers 0 to 9 before displaying it.

We declare and initialize the array as before, while initializing the individual elements in a `for` loop:

```
double A[] = new double[10];
for (int i = 0; i < A.length; i++)
    A[i] = (double)i;
```

We use explicit type casting to convert the integer loop control variable `i` into a `double` even though this conversion would take place automatically. To display the elements of the array on the screen we use a similar loop:

```
for (int i = 0; i < A.length; i++)
    System.out.println(A[i]);
```

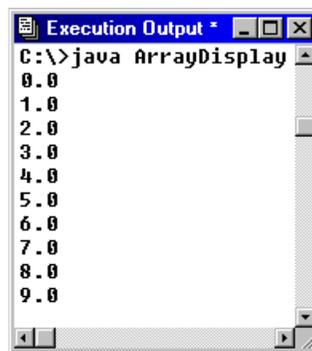


Figure 2.18: Displaying an array

Note that we have used `A.length` to control both `for` loops. If we want to display the numbers from 0 to 999 all we have to do is change the size of the array to 1000. ■

Software Engineering Tip: Arrays are almost always processed using one or more `for` loops. To control such loops you should always use the `length` field of the array, never a numeric value. That way your code adjusts automatically whenever the size of an array changes.¹⁷

Remember that the indices for an array `A` of length `N` are `0`, `1`, `2`, ..., `N-1`. In particular, statements such as `A[A.length]` will generate a runtime error.

Arrays of any type can be defined, as long as all elements in the array have the same type.

Example 2.13: Arrays of String

Use an array to store the sentence "Java is an object oriented programming language". Display that sentence and find the average number of characters per word.

Problem Analysis: To find the average number of characters per word we need to divide the total number of characters by the number of words:

$$\text{avg. chars per word} = (\text{total chars}) / (\text{number of words})$$

Program Implementation: To store the sentence we use an array of `String` called `sentence`, initialized by a comma-delimited list enclosed in curly brackets. We use a `for` loop to display the sentence, and the same loop to compute the total number of characters. Each array element is of type `String`, therefore the method `sentence[i].length()` returns the number of characters in the `i`-th word. A call to the field `sentence.length` will give us the number of words in `sentence`¹⁸:

```
public class AverageCharacters
{   public static void main(String args[])
    {   String sentence[] = {"Java ", "is ", "an ", "object ", "oriented ",
                            "programming ", "language"};

        int numChars = 0;
        for (int i = 0; i < sentence.length; i++)
        {   System.out.print(sentence[i]);
            numChars += sentence[i].length();
        }
        System.out.println();
        System.out.println("Total chars: " + numChars);
        System.out.println("Total words: " + sentence.length);
        System.out.println("Avg. Chars: " + (double)numChars / sentence.length);
    }
}
```

Arrays as Input Parameters

Arrays can also be used as input and return types for methods. Since arrays are reference parameters some special considerations apply when using them as input to methods.

¹⁷ In fact, even when initializing an array the size should not be specified as a fixed number. Instead, a suitable `int` field, variable, or named constant should be used.

¹⁸ Note the difference between `sentence.length` and `sentence[i].length()`. Since `sentence` is an array, the *field* `sentence.length` contains its size. But `sentence[i]` is of type `String`, so the *method* `sentence[i].length()` returns the number of characters in that string.

Arrays as Method Parameters

Arrays can be used as input parameters for methods as well as for return types. They are declared using empty square brackets without specifying the size.

Arrays are reference types. If they are used as input parameters, changes to the elements of the array inside a method will be reflected outside the method.

Example 2.14: Arrays as method input parameters

What are the values that will be displayed in the following program:

```
public class ArrayParameters
{
    public static void changeOrNot(int i, double x[])
    {
        i = -1;
        x[0] = -2.0;
        double y[] = x;
        y[1] = -3.0;
        double z[] = {-4, -4, -4};
        x = z;
    }
    public static void main(String args[])
    {
        int k = 1;
        double A[] = {1.0, 2.0, 3.0};
        changeOrNot(k, A);
        System.out.println("k: " + k);
        System.out.println("A[0]: " + A[0]);
        System.out.println("A[1]: " + A[1]);
        System.out.println("A[2]: " + A[2]);
    }
}
```

It is easy to explain what happens to the integer parameter `i` of `changeOrNot`: the value 1 is copied into it from `k`, and since `i` is a value parameter of basic type nothing the method does to `i` inside the method body is visible outside.

The tricky part is the array `A`. It is passed into the method `changeOrNot` and its value is copied to the array `x`. Both `A` and `x` are separate variables pointing to the same data in memory. Therefore, when `x[0]` changes to `-2`, so does `A[0]`. Then a new array `y` is declared and initialized to refer to the same data as `x`. Therefore, `x`, `y`, and `A` now share the same elements, so that when `y[1]` changes to `-3`, so do `A[1]` and `x[1]`. Finally, another array `z` is declared and initialized.

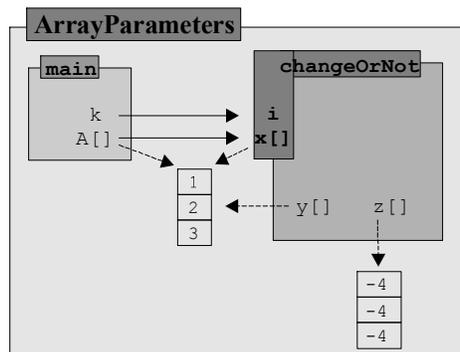


Figure 2.19: `ArrayParameters` state before changing `x` in the last line of `changeOrNot`

The value of `x` is changed so that `x` now points to the same data as `z`. That change is *not* visible outside the method, since the value of `x` was *copied* from `A` and `A` still has its original value.

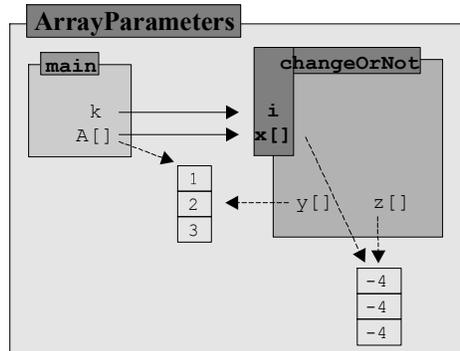


Figure 2.20: ArrayParameters state after changing `x[]`

Therefore, the printout will be:

```
k:      1
a[0]:  -2.0
a[1]:  -3.0
a[2]:   3.0
```

■

The next example uses arrays and methods in a complete program:

Example 2.15: Compound interest computation using arrays

Compute the compounded interest over 10 years if \$10,000 are deposited at a yearly interest rate of 6.5%, assuming that interest is compounded once per month (compare example 2.09). The output should consist of the interest per year, the new yearly balance, as well as the average interest amount up to that year.

This example is similar to example 2.09 but with two twists: the interest is compounded monthly and we want to know the average interest up to a given year. Such an average, incidentally, is called prefix average and is used frequently in finance.

Problem Analysis: To figure out what exactly we are supposed to do, we use a simple example that we can compute manually. Let's assume we want to invest \$100 at 20% for 3 years, compounded twice per year. Then for each of the two interest computations per year the rate is $\frac{1}{2}$ of 20%, or 10%. The computations are easy for the first year:

- Start amount: \$100
- Interest for first period: $\$100 * 10\% = \$100 * 0.1 = \$10$
- New amount after first period: $\$100 + \$10 = \$110$
- Interest for second period: $\$110 * 10\% = \$110 * 0.1 = \$11$
- Total interest for first year: $\$10 + \$11 = \$21$
- Total amount after first year: $\$100 + \$21 = \$121$

Table 2.21 shows the amounts for year 1, 2, and 3 of our simple example.

Intermediate Computation					Final Amounts	
Year	Amount	1 st Interest	New Amount	2 nd Interest	Interest	Amount

1	\$100.00	\$10.00	\$110.00	\$11.00	\$21.00	\$121.00
2	\$121.00	\$12.10	\$133.10	\$13.31	\$25.41	\$146.41
3	\$146.41	\$14.64	\$161.05	\$16.11	\$30.75	\$177.16

Table 2.21: Example of compound interest computation

- The average interest for the first year is: \$21
- The average interest up to the second year is: $(\$21 + \$25.41) / 2 = \$23.20$
- The average interest up to the third year is: $(\$21 + \$25.41 + \$30.75) / 3 = \26.72

Now it is clear that there are several steps involved:

1. Compute the yearly compound interest
2. Add the yearly interest to the balance for that year
3. Compute the average interest amounts
4. Show the results

Program Implementation: According to these four steps we need four methods:

- Method `interestPerYear` needs to know the beginning balance, the rate, and the number of periods and returns the compounded interest for one year.
- Method `adjustBalances` adjusts the balance for each year.
- Method `prefixAverage` computes the average interest amount up to a given year. It needs the interest amounts for each year to compute the prefix average. The output will be an array where the i -th element corresponds to the average interest up to year i .
- Method `showAmounts` should display the balances, interests, and average interests for all years on the screen.

We also need two arrays:

- an array to store the balances for each year
- an array to store the total interest for each year

Software Engineering Tip: Arrays are useful and – after a little experience – simple to use. They should be declared whenever three or more variables of the same type are needed that have some logical connection with each other.

We could define these arrays as fields, similar to example 2.09, but arrays are reference types. If a method modifies the array elements, the modifications *are* visible outside the method. Therefore we can use local variables declared and initialized in the standard `main` method. The arrays should be input parameters to the various methods, so that our method headers are:

```
public static double interestPerYear(double amount)
public static void adjustBalances (double amount[], double interest[])
public static double[] prefixAvg(double A[])
public static void showAmounts(double amount[], double interest[],
                               double avg[])
```

The remaining information such as the starting balance, interest rate, number of periods for yearly compounding, and the number of years will be stored in constant fields available to all methods. Our class so far looks as follows, with the `main` method completely implemented:

```
public class CompoundInterest
{   final static int    NUM_YEARS = 10;
    final static int    NUM_PERIODS = 12;
```

```

final static double RATE = 6.5;
final static double START_BALANCE = 10000;

public static double interestPerYear(double amount)
{ }
public static void adjustBalances(double amount[], double interest[])
{ }
public static double[] prefixAvg(double A[])
{ }
public static void showAmounts(double amount[], double interest[],
double avg[])
{ }
public static void main(String args[])
{ double amount[] = new double[NUM_YEARS];
  double interest[] = new double[NUM_YEARS];
  interest[0] = interestPerYear(START_BALANCE);
  amount[0] = START_BALANCE + interest[0];
  adjustBalances(amount, interest);
  showAmounts(amount, interest, prefixAvg(interest));
}
}

```

Method `interestPerYear` is similar to example 2.09 but it returns the compounded interest on the input `amount` for one year instead of adjusting the balance:

```

public static double interestPerYear(double amount)
{ double compoundInterest = 0.0;
  for (int i = 0; i < NUM_PERIODS; i++)
  { double interest = amount * RATE / 100 / NUM_PERIODS;
    amount += interest;
    compoundInterest += interest;
  }
  return compoundInterest;
}

```

Method `adjustBalances` calls on `interestPerYear` for each year to adjust the balances for all years except the first and stores the computed interest amounts in the `interest` array. The first year was already initialized in the `main` method:

```

public static void adjustBalances(double amount[], double interest[])
{ for (int year = 1; year < NUM_YEARS; year++)
  { interest[year] = interestPerYear(amount[year-1]);
    amount[year] = amount[year-1] + interest[year];
  }
}

```

The `prefixAvg` method uses a double for loop to compute the prefix averages. The outer loop goes from year 0 to `NUM_YEARS-1` year. For each value of that loop counter an inner loop computes the average of the input array up to the value of the outer loop:

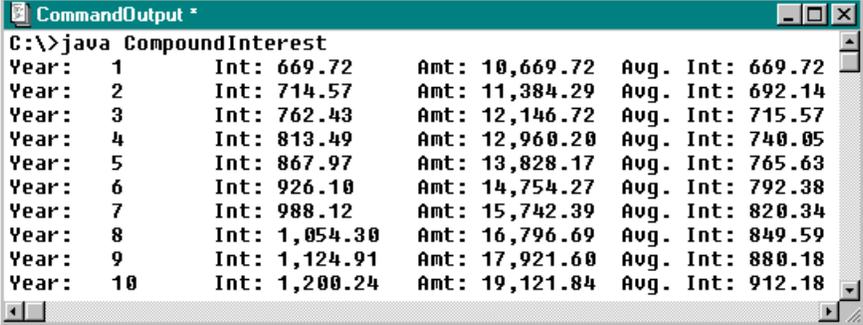
```

public static double[] prefixAvg(double A[])
{ double avg[] = new double[A.length];
  for (int year = 0; year < A.length; year++)
  { double sum = 0.0;
    for (int j = 0; j <= year; j++)
      sum += A[j];
    avg[year] = sum / (year + 1);
  }
  return avg;
}

```

Finally, `showAmounts` will display all answers in an appropriate format:

```
public static void showAmounts(double amount[], double interest[],
                               double avg[])
{
    DecimalFormat formatTool = new DecimalFormat("#,###.00");
    for (int i = 0; i < NUM_YEARS; i++)
    {
        System.out.print("Year:\t" + (i+1));
        System.out.print("\tInt: " + formatTool.format(interest[i]));
        System.out.print("\tAmt: " + formatTool.format(amount[i]));
        System.out.println("\tAvg. Int: " + formatTool.format(avg[i]));
    }
}
```



```
C:\>java CompoundInterest
Year: 1      Int: 669.72      Amt: 10,669.72  Avg. Int: 669.72
Year: 2      Int: 714.57      Amt: 11,384.29  Avg. Int: 692.14
Year: 3      Int: 762.43      Amt: 12,146.72  Avg. Int: 715.57
Year: 4      Int: 813.49      Amt: 12,960.20  Avg. Int: 740.05
Year: 5      Int: 867.97      Amt: 13,828.17  Avg. Int: 765.63
Year: 6      Int: 926.10      Amt: 14,754.27  Avg. Int: 792.38
Year: 7      Int: 988.12      Amt: 15,742.39  Avg. Int: 820.34
Year: 8      Int: 1,054.30    Amt: 16,796.69  Avg. Int: 849.59
Year: 9      Int: 1,124.91    Amt: 17,921.60  Avg. Int: 880.18
Year: 10     Int: 1,200.24    Amt: 19,121.84  Avg. Int: 912.18
```

Figure 2.22: Output of `CompoundInterest` program

Figure 2.22 shows that the interest increases year after year because the compounded balance gets larger and larger. The average interest, on the other hand, implies that from year 1 to year 10 we have received an average interest of \$912.18, which is equivalent to an average interest rate of 9.12% *without* compounding. Thus, a fixed interest rate of 6.5% over 10 years, compounded monthly, is approximately equivalent to a 9.12% fixed interest rate without compounding. ■

Now that we understand arrays as well as methods, we also understand what the header of the `main` method means:

```
public static void main(String args[])
```

declares a method called `main` that returns nothing and has as input an array of strings referred to by the local variable `args`. The value copied into `args` originates from the command line.

Command Line Input

A Java program can receive any number of parameters as input from the command line. These input parameters are automatically stored in the array `args` declared in the header of the `main` method. To pass values into a Java program¹⁹ invoke the program as follows:

```
java ClassName "input value1" "input value2" ... "input value N"
```

The field `args.length` can be used to determine the number of input strings received.

¹⁹ This feature can be used to pass user input into a program. All input parameters will be of type `String`, but they can be converted to `int` or `double` if desired (compare with the `Console` class in section 2.4).

Example 2.16: Command line input to main method

Create a program that prints out the values that were passed into it from the command line, if any, or an appropriate message if there were no command line parameters.

The variable `args` inside the `main` method automatically contains the values of the command line parameters, if any. Its `length` field contains the number of strings that were specified:

```
public class CommandLine
{
    public static void main(String args[])
    {
        if (args.length == 0)
            System.out.println("No command line arguments");
        else
        {
            System.out.println("Command line arguments were: ");
            for (int i = 0; i < args.length; i++)
                System.out.println("Argument " + (i+1) + ": " + args[i]);
        }
    }
}
```

Here are the results of two runs of this program:

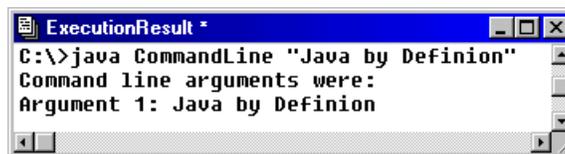


Figure 2.23: A Java program with one command line parameter

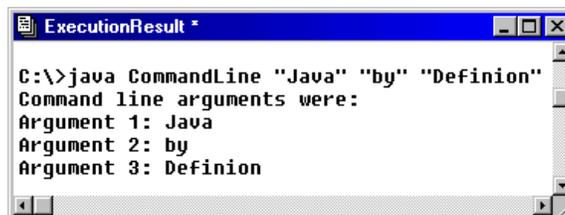


Figure 2.24: A Java program with three command line parameters

We will encounter arrays in many of our later programs and the exercises contain plenty of additional examples to gather more experience using arrays.

Higher-Dimensional Arrays

Java can create higher-dimensional arrays by considering them as "arrays of arrays". The most common type of higher a dimensional array is a two-dimensional array.²⁰

²⁰ Java can handle arrays of higher dimensions (arrays of arrays of arrays ...). We will focus on two-dimensional arrays only and leave three-dimensional arrays as an exercise.

Two-Dimensional Arrays

A two-dimensional array is an array where each element in turn is an array. Two-dimensional arrays can be represented as a table, where the size of the first array determines the number of rows and the size of the elements of the first array, which in turn are arrays, determines the number of columns²¹. Two-dimensional arrays in Java are declared and initialized using:

- *to declare:* `type tableName[][];`
- *to initialize:* `tableName = new type[numRows][numCols]`
- *to declare and initialize:* `type tableName[][] = new type[numRows][numCols];`

Row and column indices start at 0 and go up to `numRows-1` and `numCols-1`, respectively. Two-dimensional arrays can be declared and initialized implicitly by providing a comma-separated list of values in curly brackets internally grouped according to the number of rows and columns:

```
type tableName[][] = {{val1, val2, ..., valN}, ..., {val1, val2, ..., valM}};
```

Two-dimensional arrays can be imagined as tables, while in mathematics a two-dimensional array is frequently called a matrix (which should be familiar from linear algebra).

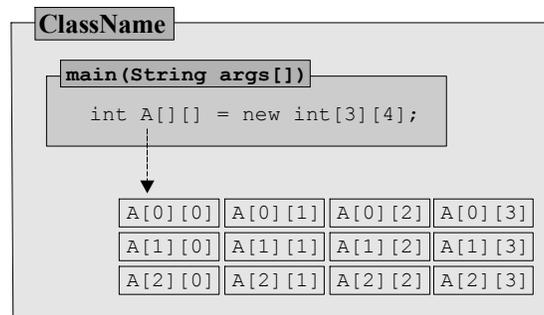


Figure 2.25: Declaring and initializing a two-dimensional array with 3 rows and 4 columns

Example 2.17: Working with two-dimensional arrays

Define a two-dimensional array (or matrix) with 4 rows and 4 columns of `double`, then create methods to display the array, to find the sum of any given row, of any given column, of the main diagonal, of the minor diagonal, and the largest number in the array.

Problem Analysis: Let's look at a two-dimensional array, or matrix, of size 4 by 4 to see if we can determine by example what we are supposed to do. The notation $a_{1,2}$ denotes an element in row 1, column 2 (row and column indices start at 0, so row 1 refers to the 2nd row, column 2 to the 3rd column):

²¹ As we will see in example 9, two-dimensional arrays do not have to be rectangular, the number of columns could be different in different rows. To declare and initialize non-rectangular two-dimensional arrays use `type arrayName[][] = new type[numRows][]` and for each row use `arrayName[row] = new type[numColsForRow]`.

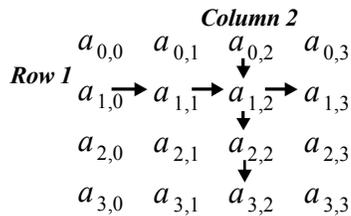


Figure 2.26(a): Matrix rows and columns

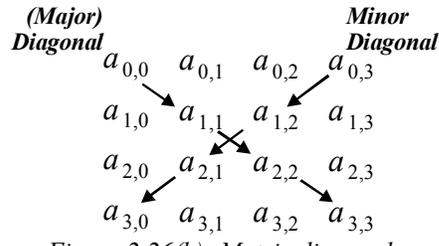


Figure 2.26(b): Matrix diagonals

Examples of row/column/diagonal summations:

- sum of row 1: $a_{1,0} + a_{1,1} + a_{1,2} + a_{1,3}$
- sum of column 2: $a_{0,2} + a_{1,2} + a_{2,2} + a_{3,2}$
- sum of diagonal: $a_{0,0} + a_{1,1} + a_{2,2} + a_{3,3}$
- sum of minor diagonal: $a_{0,3} + a_{1,2} + a_{2,1} + a_{3,0}$

Program Implementation: In Java, this two-dimension matrix is created using:

```
double A[][] = new double[4][4];
```

where the elements in A are automatically set to zero. Once we realize the patterns for finding the various sums, it is easy to come up with appropriate methods for finding the sums of rows, columns, and main diagonal:

```
public static double sumOfRow(double matrix[][], int row)
{ double sum = 0.0;
  for (int col = 0; col < matrix[0].length; col++)
    sum += matrix[row][col];
  return sum;
}
public static double sumOfCol(double matrix[][], int col)
{ double sum = 0;
  for (int row = 0; row < matrix.length; row++)
    sum += matrix[row][col];
  return sum;
}
public static double sumOfDiagonal(double matrix[][])
{ double sum = 0.0;
  for (int diag = 0; diag < matrix.length; diag++)
    sum += matrix[diag][diag];
  return sum;
}
```

The method `sumOfDiagonal` only needs the matrix as input because there is only one diagonal to sum up. To find the sum of the *minor* diagonal, i.e. the diagonal entries from the upper right to the lower left, we need one `for` loop so that inside the loop the row index goes up while the column index goes down:

```
public static double sumOfMinorDiagonal(double matrix[][])
{ double sum = 0.0;
  for (int i = 0; i < matrix.length; i++)
    sum += matrix[i][matrix.length-i-1];
  return sum;
}
```

Next, we need to find the largest value in a matrix. We need a double `for` loop: one loop to go through the rows and another one to go, for each fixed row, through each column. As we investigate each element, we compare it with what we previously found to be the largest value. If we find a larger one in the matrix, we use that as our new maximum:

```
public static double findMax(double matrix[][])
{
    double max = matrix[0][0];
    for (int row = 0; row < matrix.length; row++)
    {
        for (int col = 0; col < matrix[row].length; col++)
        {
            if (matrix[row][col] > max)
                max = matrix[row][col];
        }22
    }
    return max;
}
```

The call to `matrix[row].length` will return the size of the array in row `row`, or in table-terminology the number of columns in that row. Since all rows have the same number of columns we could have used `matrix[0].length` for all rows.

Software Engineering Tip: Use either named constants or other derived constants to define and manipulate 2-dimensional arrays. In particular, make use of the correct number of rows and of elements in each row by using `A[].length` and `A[row].length`, respectively, where `A` is a two-dimensional array and `row` an integer specifying a particular row. This code will automatically adjust to the particular size of a two-dimensional array even if it is not rectangular.

Finally, we need the code to display a matrix. We have already seen that `System.out.println` works for arrays (or for any other type), but it prints out the memory address not the array elements. Therefore, just as for one-dimensional arrays, we need to create our own output method for matrices. The first attempt goes through the matrix in a double `for` loop and displays each element:

```
public static void showMatrix(double matrix[][])
{
    for (int row = 0; row < matrix.length; row++)
        for (int col = 0; col < matrix[row].length; col++)
            System.out.println(matrix[row][col]);
}
```

This displays the entire matrix, but it puts each element in a single line. That's not what we want. If we replace `System.out.println` with `System.out.print`, all elements will appear in one line with no spacing between them. That's also not what we want. We need to insert `tab` characters between the elements of one row and a `linefeed` after each row so that the matrix will resemble a table:

```
public static void showMatrix(double matrix[][])
{
    for (int row = 0; row < matrix.length; row++)
    {
        for (int col = 0; col < matrix[row].length; col++)
            System.out.print(matrix[row][col] + "\t");
        System.out.print("\n");
    }
    System.out.print("\n");
}
```

■

²² The grouping brackets for the `for` loops are not necessary in this case but are included here for clarity.

Example 2.18: Two-dimensional array representing a table of sales figures

Create a table representing sales figures for a large company. The rows of the table should correspond to the regions *North* and *South*, and the columns should represent the categories *Toys*, *Clothing*, and *Educational*. Fill the table with random numbers, display the table, and compute the total sales for the regions and categories.

Problem Analysis: We need to generate a two-dimensional array with 2 rows and 3 columns, where the rows correspond to sales in one of the two regions and the columns to one of the three categories. We then need to display that two-dimensional array and compute the sum of the rows corresponding to the regions, and the sum of the columns corresponding to the categories.

Program Implementation: We can use the methods `sumOfRow` and `sumOfCol` as defined in example 2.17 without change, as well as the method `showMatrix` with a few minor changes. We use fields of type array of `String` as labels for the regions and category. The `main` method declares and initializes our table, which is then used as input to the various methods.

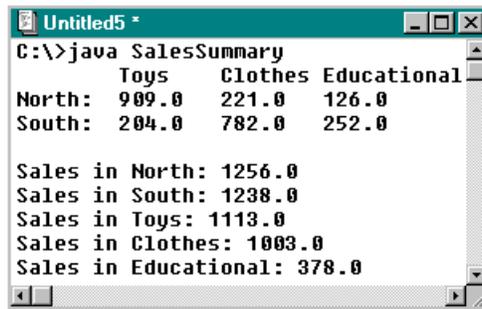
```
public class SalesSummary
{   final static String[] CATEGORIES = {"Toys", "Clothes", "Educational"};
    final static String[] REGIONS = {"North", "South"};

    public static double sumOfRow(double matrix[][], int row)
    { /* as in example 2.17, no changes */ }
    public static double sumOfCol(double matrix[][], int col)
    { /* as in example 2.17, no changes */ }
    public static void showMatrix(double matrix[][])
    {   System.out.print("\t");
        for (int i = 0; i < CATEGORIES.length; i++)
            System.out.print(CATEGORIES[i] + "\t");
        System.out.println();
        for (int row = 0; row < matrix.length; row++)
        {   System.out.print(REGIONS[row] + ":\t");
            for (int col = 0; col < matrix[row].length; col++)
                System.out.print(matrix[row][col] + "\t");
            System.out.print("\n");
        }
        System.out.print("\n");
    }
    public static void main(String args[])
    {   double table[][] = new double[REGIONS.length][CATEGORIES.length];
        for (int row = 0; row < table.length; row++)
            for (int col = 0; col < table[row].length; col++)
                table[row][col] = 1000*Math.random();

        showMatrix(table);
        for (int region = 0; region < REGIONS.length; region++)
            System.out.println("Sales in " + REGIONS[region] + ": " +
                               sumOfRow(table, region));

        for (int cat = 0; cat < CATEGORIES.length; cat++)
            System.out.println("Sales in " + CATEGORIES[cat] + ": " +
                               sumOfCol(table, cat));
    }
}
```

We made some minor changes to `showMatrix` in order to display the row and column headers appropriately. Figure 2.27 shows the results of running this program:



```

C:\>java SalesSummary
      Toys   Clothes Educational
North: 909.0  221.0   126.0
South: 204.0  782.0   252.0

Sales in North: 1256.0
Sales in South: 1238.0
Sales in Toys: 1113.0
Sales in Clothes: 1003.0
Sales in Educational: 378.0

```

Figure 2.27: Executing the SalesSummary program

Two-dimensional arrays do not necessarily have to be rectangular, the number of row elements can be different from one row to the next.

Example 2.19: Triangular array for prefix sales averages

Let's assume we have bimonthly sales figures in millions of dollars for a corporation. We want to generate a (prefix) sales report, listing in each row the total and average sales up to that month and the individual sales figures for those months. Generate that report, using a two-dimensional triangular array for the bimonthly data, and another array for the sum and average figures.

Problem Analysis: Suppose the sales figures for Feb, Apr, and May are 30, 20, and 40, respectively. We want our program to generate a report that looks similar to table 2.28:

	Sum	Avg	Feb	Apr	May
Feb	30	30	30		
Apr	50	25	30	20	
May	90	30	30	20	40

Table 2.28: Example sales report with prefix averages

There are at least three distinct tasks involved:

- generate the triangular array for sales data per month
- compute the sums and totals based on that triangular array
- display the final report

Program Implementation: Of course that means we will use three methods. To make our program flexible, we first define a field `MONTHS` of type array of strings that specifies the valid months for our sales data. We then create the method to generate the triangular table. Assuming `salesFigures` is a one-dimensional `int` array containing our sales figures, the method takes as input the `salesFigures` array and produces as output a triangular table of sales data (see figure 2.30). The table has as many rows as there are months, but each row has a different number of columns. We will declare the table as:

```
int table[][] = new int[MONTHS.length][];
```

specifying only the number of rows. We generate the number of columns in a `for` loop, then use another `for` loop to fill the table with the proper numbers from `salesFigures`:

```
public static int[][] generateTable(int salesFigures[])
{
    int table[][] = new int[MONTHS.length][];
    for (int row = 0; row < salesFigures.length; row++)

```

```

        table[row] = new int[row+1];

    for (int row = 0; row < table.length; row++)
        for (int col = 0; col < table[row].length; col++)
            table[row][col] = salesFigures[col];
    return table;
}

```

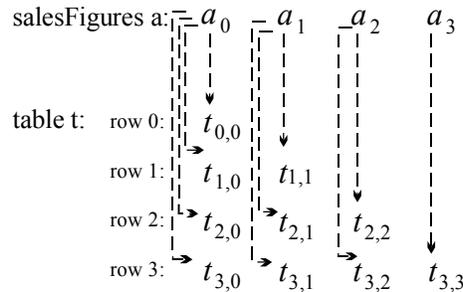


Figure 2.29: Converting a one-dimensional array into a triangular matrix

The second method to compute the summary for our data takes as input the table produced by `generateTable` and computes the totals and averages per row. It returns a two-dimensional array where the first column contains totals and the second the averages per row. Again a double `for` loop will do the trick:

```

public static int[][] computeSummary(int table[][])
{
    int summary[][] = new int[MONTHS.length][2];
    for (int row = 0; row < table.length; row++)
    {
        summary[row][0] = 0;
        for (int col = 0; col < table[row].length; col++)
            summary[row][0] += table[row][col];
        summary[row][1] = summary[row][0] / table[row].length;
    }
    return summary;
}

```

The last method takes as input the table created by `generateTable` and the summary from `computeSummary` to produce the desired report. It first creates a header row for the report, consisting of the strings "Sum", "Avg", and the months in the `MONTHS` array. Next we display the data for each row: first we print the month again, then the total and the average, and finally the data from the table in a `for` loop. We will add a vertical line symbol "|" at the appropriate places to make the table more readable.

```

public static void showReport(int table[][], int summary[][])
{
    System.out.print(" \t| Sum\tAverage\t| ");
    for (int i = 0; i < MONTHS.length; i++)
        System.out.print(MONTHS[i] + "\t");
    System.out.println();
    for (int row = 0; row < table.length; row++)
    {
        System.out.print(MONTHS[row] + "\t| ");
        System.out.print(summary[row][0] + "\t" + summary[row][1] + "\t| ");
        for (int col = 0; col < table[row].length; col++)
            System.out.print(table[row][col] + "\t");
        System.out.println();
    }
}

```

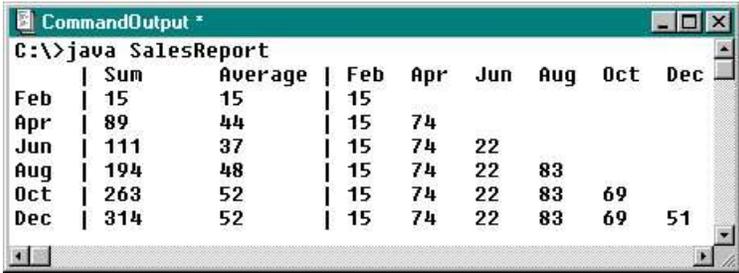
With these methods doing all the work, the job of the `main` method is easy. It generates some random sales data and calls on the above methods in the proper order to compute everything and display the report:

```
public class SalesFigures
{   static String MONTHS[] = {"Feb", "Apr", "Jun", "Aug", "Oct", "Dec"};

    public static int[][] generateTable(int salesFigures[])
    { /* code as above */ }
    public static int[][] computeSummary(int table[][])
    { /* code as above */ }
    public static void showReport(int table[][], int summary[][])
    { /* code as above */ }
    public static void main(String args[])
    {   int salesFigures[] = new int[MONTHS.length];
        for (int i = 0; i < salesFigures.length; i++)
            salesFigures[i] = (int)(100 * Math.random());

        int table[][] = generateTable(salesFigures);
        int summary[][] = computeSummary(table);
        showReport(table, summary);
    }
}
```

This program produces a report comparable to table 2.28 where you can see which months' sales figures were better or worse than average together with the actual numbers per period.



	Sum	Average	Feb	Apr	Jun	Aug	Oct	Dec
Feb	15	15	15					
Apr	89	44	15	74				
Jun	111	37	15	74	22			
Aug	194	48	15	74	22	83		
Oct	263	52	15	74	22	83	69	
Dec	314	52	15	74	22	83	69	51

Figure 2.30: SalesReport output

The averages up to Feb. and Jun., for example, are lower than the rest because the (random) sales figures for those months were disappointing. ■

We will revisit two-dimensional arrays in the last section of this chapter where we present a program that can solve linear equations by matrix manipulations. The next section introduces guidelines for adding comments and documentation to your code and for testing and debugging programs.

2.4. Documenting, Testing, and Keyboard Input

It is important in every programming language to document and test your code appropriately. In object-oriented languages such as Java, good documentation and testing plays an even more important role. Java allows you to easily create objects that can be reused without modification in any number of programs. Unless the capabilities and shortcomings of those objects are well documented and tested, they are (almost) useless.

To help document your code the JDK provides not only the capabilities of inserting comments into your source code – called implementation comments – but also a tool called `javadoc` to automatically generate complete, formatted documentation for your programs and objects that is indexed and cross-referenced. This documentation is stored in HTML format so that it can be viewed with any web browser and it does not contain your actual source code.

Implementation and Documentation Comments

Implementation Comments

You can embed two types of implementation comments in your source code to document complicated sections:

- *A comment that applies from the starting point to the end of line is used to insert brief comments about your code. Such a comment is started by the double-slash symbols `//`*
- *Comments that apply from a given starting point to a particular endpoint are used to insert larger notes about your code. Such comments start where you put the slash-star symbols `/*` and end where you put the star-slash symbols `*/`*

The compiler ignores both types of comments and they do not increase the size of the class file.

Example 2.20: Proper and improper use of comments

Look at the code below and identify the comments. Then decide which comments are not necessary because the code is clear, and which comments indicate that the corresponding code should be rewritten.

```
/*
   This code finds the length of the hypotenuse of a right
   triangle
*/
double x = 10.0;    // x stands for the base;
double y = 20.0;    // y stands for the height;
/*
   Now computing the length by taking the square root of the
   sum of x*x and y*y
*/
double z = Math.sqrt(x*x + y*y);
```

The first comment introduces the code. It is always a good idea to start longer segments of code with a brief introductory comment explaining what is about to happen. Next, two variables `x` and `y` are declared. Since their names are non-descript, comments are used to clarify their meaning. That, of course, is a bad idea. Instead of using comments we should choose more appropriate names for the variables, such as `base` and `height`. Finally, another comment describes the computation that follows. That comment is redundant. It describes exactly what the code says and there's no need to restate the obvious. The same code, properly written, could look as follows:

```
/*
 * This code finds the length of the hypotenuse of a right
 * triangle given the base and height
*/
```

```
*/
double base = 10;
double height = 20;
double hypotenuse = Math.sqrt(base*base + height*height);
```

While comments are certainly useful, much of your code should be as clear and self-evident as possible. You should not use comments to clarify code when you could instead write clearer code. Writing the best and most readable code should always be your first goal.

Software Engineering Tip: Do not substitute comments over clear programming code. To ensure that your source code is as clear as possible, follow these guidelines:

- Use names for variables and fields that clearly signify the meaning of the stored data.
- Break up large tasks into small subtasks using methods so each subtask accomplishes a well-defined, small piece of the entire task.
- Name all methods so that their names clearly indicate what they accomplish.
- Try to name variables, fields, and methods so that they produce "English-like" sentences when used according to the Java language rules.

Occasionally it may be difficult to decide on an appropriate name for a method. That often means your method tries to accomplish too much and you should try to break it up into more subtasks until their meaning – and therefore their names – become clear.

Example 2.21: Proper use of comments

Take a look at the code below, which reverses a String, removes any leading pound symbols #, and converts the string to uppercase letters. Are the comments clear? Is there anything you would change? If so, change it.

```
/*
   This method reverses the input string, removes any leading
   pound symbols, and converts all characters to their uppercase
   version, then returns the new, converted string.
*/
public static String changeString(String s)
{ // First, reversing the string
  String changedS = "";
  for (int i = s.length() - 1; i >= 0; i--)
    changedS += s.charAt(i);

  // Next, removing any leading pound signs
  while ((changedS.length() > 0) && (changedS.charAt(0) == '#'))
    changedS = changedS.substring(1, changedS.length());

  // Finally, converting to upper-case characters
  changedS = changedS.toUpperCase();

  // and returning the converted String
  return changedS;
}
```

While this method does work and has proper comments explaining what it is doing, the name of the method `changeString` is too vague. That is in fact the reason why the introductory comment is necessary. Further comments were used to explain various subsections of the code. This is not at all good programming style. For example, try to figure out, from the comments alone, what this method

will do to the string "# Otto". Looking at the comments only one might be tempted to say that the method returns the string "OTTO" (leading # removed, reversed, and capitalized). However, the method instead returns "OTTO #" because *first* it reverses the string, and *then* it removes leading # symbols.

Instead of the above method with comments, the following code without comments is clearer and more flexible:

```
public static String reverseString(String s)
{ String reverseS = "";
  for (int i = s.length() - 1; i >= 0; i--)
    reverseS += s.charAt(i);
  return reverseS;
}
public static String removeLeadingChars(String s, char removeThis)
{ while ((s.length() > 0) && (s.charAt(0) == removeThis))
  s = s.substring(1, s.length());
  return s;
}
```

We have broken up the original method into two methods whose names are descriptive enough that we do not need comments. In addition, we managed to enhance the method to remove the leading # symbols so that it can remove *any* leading character. We did not create code for converting the string to uppercase characters because a `String` already has that capability built-in as one of its methods. To convert, say, "# Otto" we would use our methods as follows:

```
String s = "# Otto";
String convert = removeLeadingChars(s, '#');
convert = reverseString(convert);
convert = convert.toUpperCase();
```

or, if we wanted to, we could combine the last three statements into one:

```
String s = "# Otto";
String convert = reverseString(removeLeadingChars(s, '#')).toUpperCase();
```

which again raises the question as to which code is more readable (left up to you this time). ■

Java also supports comments called documentation comments, which can be used by the JDK tool `javadoc` to automatically generate HTML documentation for your classes²³.

Documentation Comments

Java allows you to add documentation comments to your source code that can be extracted and formatted by the `javadoc` utility program that is part of the JDK package. This utility program will extract specifically tagged comment lines from your source code, process, format, index, cross-link them, and produce a sequence of linked HTML documents. Follow these rules to insert documentation comments:

- *documentation comments begin with the slash-star-star symbols `/**` and end with the star-slash symbols `*/`*

²³ If you are creating classes that can be reused in many situations, you really should use documentation comments to let `javadoc` create professional documentation for you.

- *documentation comments can include HTML formatting directives (except headline tags)*
- *documentation comments must appear directly before the class name, field, or method they refer to*
- *documentation comments can contain special directives standing in a line by themselves:*
 - `@author Name` *to indicate the author of the code*
 - `@version Number/Date` *to indicate the version and perhaps date of the code*
 - `@see class#references` *for links to other documentation comments*
- *documentation comments referring to a method can include the directives:*
 - `@return rType` *to indicate the return type of the method*
 - `@exception exType` *to indicate any exception this method throws*

Example 2.22: Using javadoc comments

We have already created the methods `reverseString` and `removeLeadingChars`. Use javadoc comments to document a program that includes these methods.

Since we have already created these methods, here is the code embedding them into a complete program with javadoc comments:

```

/** This is a test program to test the methods <code>reverseString</code>
    and <code>removeLeadingChars</code>.

    @author Bert G. Wachsmuth
    @version 98/03/10
 */
public class StringAndComments
{
    /** This method reverses the input string, i.e. the input string is
        returned character by character in reverse order.

        @return String
    */
    public static String reverseString(String s)
    {
        /* code as before */
    }

    /** This method removes all leading characters from the input
        string s. The character to be removed is determined by
        the input parameter <code>removeThis</code>

        @return String
    */
    public static String removeLeadingChars(String s, char removeThis)
    {
        /* code as before */
    }

    /** The main method tests <code>removeLeadingChars</code> and
        <code>reverseString</code> to see if they work correctly.

        @see StringAndComments#removeLeadingChars(String, char)
        @see StringAndComments#reverseString(String)
    */
    public static void main(String args[])
    {
        String s = "# Otto";
        String uncommented = removeLeadingChars(s,'#');
        String upper = reverseString(uncommented).toUpperCase();
        if (s.toUpperCase().equals(upper))
            System.out.println("String is a palindrome");
    }
}

```

The first `javadoc` comment describes the overall purpose of the class or program. The comment includes the HTML tags `<code>` and `</code>` to properly format code segments, the name of the author, and the version or date of the code.

Then the two methods each have a `javadoc` comment, describing briefly what they do (but not *how* they accomplish it). In particular, each method returns a `String` as indicated by the `@return` tag.

Finally, the `main` method is described. Its only purpose is to test the previous methods, so a reference is made to them via the appropriate `@see` tags. The keyword after the `@see` tag is the current class name, followed by a pound sign `#`, followed by the name of the methods with their input types but without variable names.



Software Engineering Tip: A proper program or class should always include comments. At the very least:

- Use `javadoc` comments to give a brief description of the overall purpose of the class. The first sentence in that description will be used for a class summary and should adequately and concisely describe the class.
- Include the name of the author via the `@author` tag
- Include the date or version of the code via the `@version` tag.
- Use `javadoc` comments to give a brief description of all methods in a class. Specify the meaning of all input parameters and what the methods return. If a method has particular requirements on its input parameters, specify them as well.

Much of the code listed in this textbook does not include comments because the code is discussed in detail. Your own code, on the other hand, should religiously follow the above guidelines.

Something else needs to be done to produce the final HTML documentation or we could have used source code comments instead of `javadoc` comments.

The `javadoc` Tool

The `javadoc` tool is a program that is part of the JDK. It is used to analyze and extract documentation comments from Java source documents. It produces a well-formatted, cross-linked, and indexed documentation in HTML format. To use the `javadoc` tool, type from the command line (similar to invoking the Java compiler) the line

```
 javadoc [options] SourceCodeFile.java
```

where wildcard characters are allowed for the file name. Some of the useful options include:

```
-private  show all classes and members
-version  include @version tags (omitted by default)
-author   include @author tags (omitted by default)
```

The javadoc tool produces by default one HTML file per Java source file as well as a class hierarchy file `overview-tree.html` and an index file `index.html`.²⁴

Example 2.23: Using the javadoc tool

In example 2.22 we created a program including javadoc comments to test some string conversion methods. Use the javadoc tool to produce the documentation in HTML format. Then use a web browser to view the HTML document(s) generated by javadoc.

In example 2.22 we listed the source code, which should be saved as `StringAndComments.java`. To generate the HTML documentation, we could type, at the command prompt:

```
javadoc StringAndComments.java
```

To include the author and version information we add the options `-version` and `-author`:

```
javadoc -author -version StringAndComments.java
```

Several files will be created, including `StringsAndComments.html`, which is shown in figure 2.31²⁵:

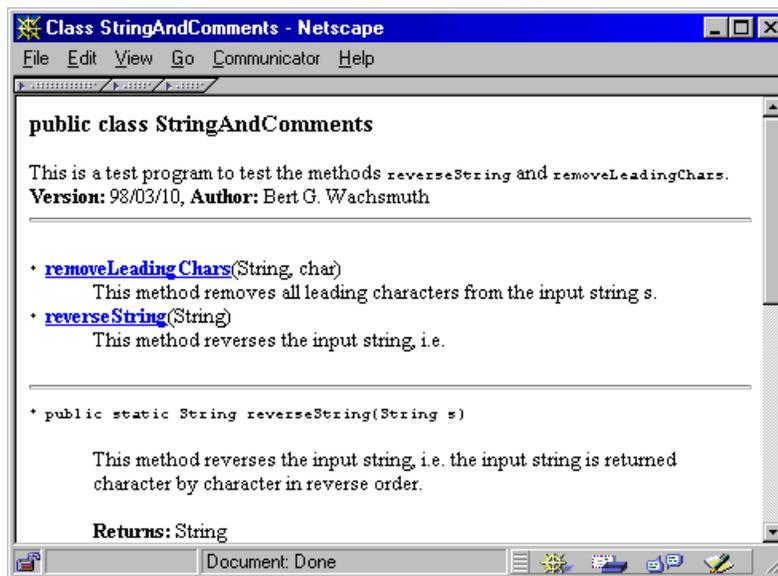


Figure 2.31: Web browser showing HTML documentation created by javadoc

Of course, javadoc can only produce the HTML text according to your comments. If you include carefully phrased javadoc comments in your source code, the javadoc tool can quickly and efficiently generate nice, concise, and useful documentation of your code without much work.

²⁴ The names of the files produced by javadoc vary depending on the version of javadoc. There will be at least one file name `ClassFile.html` per class containing documentation comments.

²⁵ In the alphabetical list of methods the sentence describing `reverseString` is cut-off after the "i.e.", because javadoc (incorrectly) interprets `period + space` as the end of the sentence, and only the first sentence is used for the summary. The full text of our javadoc comments appears later in the HTML document.

The Console Class for Keyboard Input

Here is an example of a class that extends Java's capabilities to read input from the keyboard. The actual code may be difficult to understand, but you should be able to look at the HTML document produced by javadoc and be able to use the class even though you may not understand how the code works.

Example 2.24: The Console class for simplified user input

The following source code for a Java class simplifies user input²⁶. It is not important that you understand the code that makes up this class, or even what a class is. Instead, run the javadoc utility program on the source code and look at the resulting HTML file(s). Then create a small program using this class according to that documentation.

```
import java.io.*;
/** The <B>Console</B> class provides methods that allow easy reading of the basic
    data types <TT>double</TT>, <TT>int</TT>, and <TT>String</TT> from standard input.
    <I>Must be placed in the same directory as the source code using it.</I><P>

    <B>Usage Example:</B>
    <PRE>
    System.out.println("Enter a double number: ");
    double x = Console.readDouble();
    System.out.println("You have entered: " + x);
    </PRE>

    @author Bert G. Wachsmuth
    @version 98/03/10
*/
public class Console
{ /** Reads a single double from the keyboard. Stops execution in case of error.
    @return double */
    public static double readDouble()
    { try
      { return Double.valueOf(readString().trim()).doubleValue(); }
      catch(NumberFormatException ne)
      { System.err.println("Console.readDouble: Not a double...");
        System.exit(-1);
        return 0.0;
      }
    }
    /** Reads a single int from the keyboard. Stops execution in case of error.
    @return int */
    public static int readInt()
    { try
      { return Integer.valueOf(readString().trim()).intValue(); }
      catch(NumberFormatException ne)
      { System.err.println("Console.readInt: Not an integer...");
        System.exit(-1);
        return -1;
      }
    }
    /** Reads a String from the keyboard until RETURN or ENTER key is pressed.
    @return String */
    public static String readString()
    { String string = new String();
```

²⁶ Java has, unfortunately, poor support for reading the basic data types from the keyboard in non-GUI (non-graphical) programs. This class provides some help and will be used in many future examples and exercises.

```

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
try
{
    string = in.readLine();
}
catch(IOException e)
{
    System.out.println("Console.readString: Unknown error...");
    System.exit(-1);
}
return string;
}
}

```

If you type this file exactly as shown and save it as `Console.java`, you can process it with `javadoc` to generate documentation similar to the following²⁷:

Class Console
The Console class consists of static methods that allow easy reading of the basic data types <code>double</code> , <code>int</code> , and <code>String</code> from standard input. <i>Must be placed in the same directory containing the source code using it.</i>
Usage Example:
<pre> System.out.println("Enter a double number: "); double x = Console.readDouble(); System.out.println("You have entered: " + x); </pre>
Methods:
public static double readDouble() Reads a single double from the keyboard. Stops execution in case of error.
public static int readInt() Reads a single int from the keyboard. Stops execution in case of error.
public static String readString() Reads a String from the keyboard until RETURN or ENTER key is pressed.

Figure 2.32: javadoc documentation for `Console` class

Based on this documentation we can use this extension of Java's capabilities as follows:

```

public class ConsoleTest
{
    public static void main(String args[])
    {
        System.out.print("Enter integer: ");
        int i = Console.readInt();
        System.out.print("Enter double: ");
        double x = Console.readDouble();
        System.out.println("Integer entered: " + i);
        System.out.println("Double entered: " + x);
    }
}

```

This example *will* compile as long as the file `Console.java` has been typed exactly as shown above and was saved in the same directory as `ConsoleTest.java`. Figure 2.33 shows the output of this class when executed twice, once entering the correct type and then entering a double instead of an int.



Figure 2.33: Valid and invalid use of `Console` class methods

²⁷ To conserve space, only the important elements of the formatted `Console.html` document are shown here.



This class and its method can be used in every program. It is a rather simplistic way of obtaining user input²⁸, but it will suffice until we introduce various graphic user interface elements in chapter 4 that allow for more convenient user interaction.

Example 2.25: Computing Exam Average and Standard Deviation

Create a simple grade-book program to help teachers evaluate the performance of their students. The program should store grades in an array of double and compute their average and standard deviation. The grades should be entered using the `Console` class.

Problem Analysis: We need to know the formulas for the average and standard deviation.

- The average is the sum of all numbers, divided by the number of numbers. If x is an array of size N , the formula for the average M is:

$$M = (x[0] + x[1] + \dots + x[N-1]) / N$$

- The standard deviation s is the square root of the variance v . The variance measures how far the grades deviate from the average. If x is an array of size N with average M , the formulas for the variance v and standard deviation s are:

$$v = ((x[0] - M)^2 + (x[1] - M)^2 + \dots + (x[N-1] - M)^2) / (N-1) \quad s = \text{Math.sqrt}(v);$$

We can divide our program into 4 tasks:

- Obtain the grades from the user and store them in an array
- Compute the average
- Compute the variance (from which it is easy to compute the standard deviation)
- Display the results

Program Implementation: We clearly need 4 methods:

- `getGrades` uses as input the number of grades to obtain, asks the user to enter each grade, and returns an array of double.
- `findAverage` uses the grade array as input and returns the average
- `findVariance` uses the grade array and the average as input and returns the variance
- `showResults` uses the average and variance as input and displays the results

Here is the implementation of these methods:

```
public static double[] getGrades(int N)
{ double grades[] = new double[N];
  for (int i = 0; i < grades.length; i++)
  { System.out.print("Enter grade " + (i+1) + ": ");
    grades[i] = Console.readDouble();
  }
  return grades;
}
```

²⁸ In chapter five we will describe Java's general "try-catch" mechanism for controlling errors. We could then improve the `Console` class to handle errors more conveniently. The current version simply stops a program as soon as an input error occurs, which is certainly *not* what a good program should do.

```
public static double findAverage(double grades[])
{ double sum = 0.0;
  for (int i = 0; i < grades.length; i++)
    sum += grades[i];
  return sum / grades.length;
}

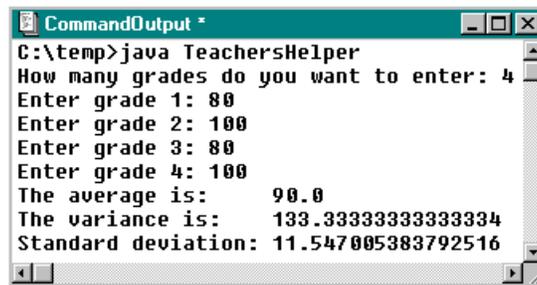
public static double findVariance(double grades[], double average)
{ double sum = 0.0;
  for (int i = 0; i < grades.length; i++)
    sum += (grades[i] - average)*(grades[i] - average);
  return sum / (grades.length - 1);
}

public static void showResults(double average, double variance)
{ System.out.println("The average is:      " + average);
  System.out.println("The variance is:     " + variance);
  System.out.println("Standard deviation: " + Math.sqrt(variance));
}
```

With these functions the standard `main` method could look as follows:

```
public static void main(String args[])
{ System.out.print("How many grades do you want to enter: ");
  int numGrades = Console.readInt();
  double grades[] = getGrades(numGrades);
  double average = findAverage(grades);
  double variance = findVariance(grades, average);
  showResults(average, variance);
}
```

Figure 2.34 shows a sample run of a program containing these methods:



```
CommandOutput *
C:\temp>java TeachersHelper
How many grades do you want to enter: 4
Enter grade 1: 80
Enter grade 2: 100
Enter grade 3: 80
Enter grade 4: 100
The average is: 90.0
The variance is: 133.33333333333334
Standard deviation: 11.547005383792516
```

Figure 2.34: Computing average and standard deviation of an array

Testing and Debugging Code

Testing your code is of course essential to create good programs and methods. A program that compiles without errors does not mean that it works correctly, and a program that works correctly in a few cases does not mean that it will work correctly in every case. A program that works correctly must contain instructions for how to use it, and a program that does not work as advertised is useless, no matter how nicely it is documented.

Error Types

In any programming language errors in source code can be classified into four types:

- *Syntactical errors:* code that is not written according to the rules of the Java language
- *Grammatical errors:* code that uses unknown language elements
- *Runtime errors:* code that produces an error while executing
- *Logical errors:* code that does not produce the results it was meant to produce

The compiler can detect syntactical and grammatical errors, while runtime and logical errors must be detected by thorough testing and program analysis.

No matter how carefully you create a method or program, it is highly likely that it initially contains one or more errors. Few programmers can create a correct method or program on their first try. In our next example we will see several tricks that can be used to develop a method or program from a "first try" to a final and documented version.

Example 2.26: Error types and debugging tips

Below is a method to replace all occurrences of the string `old` in string `s` with the string `with` but it contains mistakes. Classify and fix the mistakes to produce a working and properly documented version of this method.

```

1      public class TestAndDebug
2      {  public static String replace(String s, String old, String with)
3          {  String newString = "";
4             String remainder = s;
5             if (remainder.indexOf(old) > 0)
6                 {  int start = remainder.indexOf(old);
7                    int end   = start + old.length + 1;
8                    newString = remainder.substring(0, start) + with;
9                    remainder = remainder.substring(end, remainder.length);
10                }
11            return newString + remainder;
12        }
13    }

```

Step 1: We determine by example how the method is trying to accomplish its goal without worrying about details or errors. If the input values are `s = "reclining"`, `old = "clin"`, and `with = "member"`, the method seems to work as follows:

- In line 3 and 4 variables are initialized: `newString = ""` and `remainder = "reclining"`
- Line 5 checks if `old` is part of `remainder`, i.e. if `"clin"` is contained in `"reclining"`
- If so, line 6 sets `start` to the starting index of `old` inside `remainder` and line 7 sets `end` to one more than the ending index of `old` inside `remainder`
- Line 8 resets `newString` to the concatenation of the part of `remainder` up to `old` and `with` so that it should have the value of `"re" + "member"`
- Line 9 resets `remainder` to the part of `remainder` after `old` so that it should have the value of `"ing"`
- Line 11 returns `newString + remainder`, i.e. `"remember" + "ing"`

Step 2: We use the compiler to determine any syntactical and grammatical errors in the code²⁹:

²⁹ If the method was not part of a class we can embed it in a simple test class so that we can compile it.

```

C:\>javac TestAndDebug.java
TestAndDebug:5: Method indexOf(String) not found in class String.
    if (remainder.indexOf(old) > 0)

TestAndDebug:6: Method indexOf(String) not found in class String.
    { int start = remainder.indexOf(old);

TestAndDebug:7: Attempt to reference method length in class String as an instance variable
    int end = start + old.length + 1;

TestAndDebug:8: Method subString(int, int) not found in class String.
    newString = remainder.subString(0, start) + with;

TestAndDebug:9: Attempt to reference method length in class String as an instance variable
    remainder = remainder.subString(end, remainder.length);

5 errors
  
```

Figure 2.35: Error messages when compiling TestAndDebug

The compiler finds the following errors:

- Line 5 and 6: method `indexOf` is misspelled. It should be `indexOf` (grammar error)
- Line 7 and 9: `length` is used without parenthesis. It should be `length()` (syntax error)
- Line 8 (and 9): method `subString` is misspelled. It should be `substring` (grammar error)

Here is the new version of our method with all syntactical and grammatical mistakes fixed. It will compile without errors (which does not mean that our method will work correctly):

```

public static String replace(String s, String old, String with)
{ String newString = "";
  String remainder = s;
  if (remainder.indexOf(old) > 0)
  { int start = remainder.indexOf(old);
    int end = start + old.length() + 1;
    newString = remainder.substring(0, start) + with;
    remainder = remainder.substring(end, remainder.length());
  }
  return newString + remainder;
}
  
```

Step 3: We execute our method in a few test cases to see if it produces the desired results. To do that we add a standard `main` method to our class, which calls `replace` as follows:

```

public static void main(String args[])
{ System.out.println("Test: " + replace("reclining","clin","member"));
  System.out.println("Front: " + replace("reclining","r","R"));
  System.out.println("End: " + replace("reclining","ing","er"));
}
  
```

The results of executing this class are shown in figure 2.36:

```

C:\temp>java TestAndDebug
Test: rememberng
Front: reclining
java.lang.StringIndexOutOfBoundsException: String index out of range: -1
    at java.lang.String.substring(String.java:1492)
    at TestAndDebug.replace(TestAndDebug.java:10)
    at TestAndDebug.main(TestAndDebug.java:17)
Exception in thread "main"
  
```

Figure 2.36: Runtime error message when executing TestAndDebug

- The first test results in the string "rememberng" instead of "remembering" (logical mistake).

- The second test does not change the string at all (logical mistake).
- The third test results in an `StringIndexOutOfBoundsException` (runtime error).

To find logical or runtime errors the compiler can not help us directly but:

- we can insert `System.out.println` statements to look inside the method as it executes
- we can temporarily comment out some code to zoom in on the code we want to analyze.

We do not indent our temporary `System.out.println` statements so that we can easily remove them later:

```
public class TestAndDebug
{   public static String replace(String s, String old, String with)
    {   String newString = "";
        String remainder = s;
        System.out.println("s = " + s + ", old = " + old + ", with = " + with);
        if (remainder.indexOf(old) > 0)
        {   int start = remainder.indexOf(old);
            int end   = start + old.length() + 1;
            System.out.println("1st: " + remainder.substring(0, start));
            System.out.println("3rd: " + remainder.substring(end, remainder.length()));
            newString = remainder.substring(0, start) + with;
            remainder = remainder.substring(end, remainder.length());
        }
        return newString + remainder;
    }
    public static void main(String args[])
    {   System.out.println("Test: " + replace("reclining", "clin", "member"));
        System.out.println("Front: " + replace("reclining", "r", "R"));
        // System.out.println("End: " + replace("reclining", "ing", "er"));
    }
}
```

Running the modified class we get additional information about the inner workings of `replace`:

```
CommandOutput *
C:\temp>java TestAndDebug
s = reclining, old = clin, with = member
1st: re
3rd: ng
Test: rememberng
s = reclining, old = r, with = R
Front: reclining
```

Figure 2.37: Executing `TestAndDebug` containing temporary output statements

In the first test the 3rd part of the string is not correct. It should be "ing" instead of "ng", which implies that the value of `end` is incorrect. The second test does not show the values of the 1st and 3rd part because the test in the `if` statement returns `false`. We therefore make the following changes:

Old Line	New Line
<code>int end = start + old.length() + 1;</code>	<code>int end = start + old.length();</code>
<code>if (remainder.indexOf(old) > 0)</code>	<code>if (remainder.indexOf(old) >= 0)</code>

The third test that resulted in a runtime error did not execute because we commented it out. Since we changed our method we remove the comments and recompile and execute the class. Our `replace` method now executes correctly in all three test.

Step 4: We need to ensure that we really tested all possible situations:

- Our method should replace all occurrences of a substring with another but we only tried to replace *one* occurrence with another.
- What will our method do if the input strings `with` and `old` are empty strings?

Therefore, we will execute more tests, using the following `main` method:

```
public static void main(String args[])
{ System.out.println("No replace: " + replace("bake","coke", "pepsi"));
  System.out.println("Repeated: " + replace("Run Jane Run","Run","Go"));
  System.out.println("Replace with empty: " + replace("rerun","re",""));
  System.out.println("Replace empty: " + replace("run","","re"));
}
```

Running this program (see figure 2.38) will reveal two additional logical mistakes of our method:

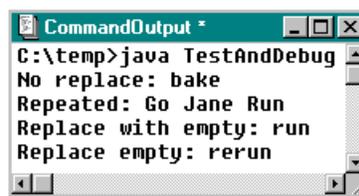


Figure 2.38: Executing `TestAndDebug` with additional logical mistakes revealed

- The method performs only one replacement and does not change all occurrences of `old`.
- The method replaces an empty string even though that should not happen.

To perform repeated substitutions we change the `if` statement to a `while` statement. Each time a replacement is made we want to append the result to `newString`, so we also change the line that resets `newString` (see below). To avoid replacing the empty string we add a second test to the `while` statement. Our final version of `replace`, which will pass all tests, is therefore:

```
public static String replace(String s, String old, String with)
{ String newString = "";
  String remainder = s;
  while ((remainder.indexOf(old) >= 0) && (!old.equals("")))
  { int start = remainder.indexOf(old);
    int end = start + old.length();
    newString += remainder.substring(0, start) + with;
    remainder = remainder.substring(end, remainder.length());
  }
  return newString + remainder;
}
```

Step 5: We add proper documentation comments to our working method:

```
/** Replaces all occurrences of <code>old</code> in string <code>s</code>
    with the string <code>with</code>.

    @author Bert G. Wachsmuth
    @version 1.0 (12/99)
 */
public static String replace(String s, String old, String with)
{ /* as before */ }
```

Example 2.26 shows several tricks that can be used to create a working version of a method or program. They can be summarized as follows:

Software Engineering Tip: Testing a method or program is important to make sure it works correctly.

- Test your method in one or more simple cases and verify the answers
- Test your method in all extreme cases and verify the answers
- Test your method when the input values are not what your method usually expects

As tips to find and correct errors in your method or program:

- Step through a method or program manually, using pencil and paper
- Embed your method(s) in a class, if necessary, and compile it to find all syntactical and grammatical errors. Use a `main` method to test all methods in your class.
- Use frequent `System.out.println` statements to display what happens during execution. Do not indent these statements so you can quickly spot and remove them again.
- Use comments to temporarily "comment-out" section of your code, either multi-line comments `/*` and `*/` for large sections or single-line comments `//` for one or two lines
- Use documentation comments to clearly state the purpose of your method and/or programs, and describe any conditions for input parameters that must be met.

2.5. Wrapper Classes for Basic Data Types

We now have a good idea about the basic elements of the Java language, but we hardly touched on the object-oriented principles that are at the heart of the language. We will examine those principles in detail in the next chapter(s) but we have already seen an example of their usefulness: the `String` and `StringBuffer` types, or more accurately, the `String` and `StringBuffer` classes. These types are different from our basic types because every variable of type `String` or `StringBuffer` contains useful operations in addition to data.

Unfortunately, the basic types `double`, `int`, `char`, and `boolean` do not have comparable operations. Therefore Java includes special "wrapper" classes for the basic types that can store data and "empower" them with operations that can be accessed via the dot operator. In this section we will introduce some of these wrapper classes and list their most important operations to start familiarizing ourselves with classes.³⁰

Java includes wrapper classes to enhance the basic data types with appropriate operations:

- All wrapper classes have names related to a basic type name, starting with a capital letter.
- All wrapper classes define reference types.
- Wrapper classes can store and return the basic types.
- Wrapper classes contain useful methods that are accessed using the dot operator.

³⁰ After finishing chapter 3 you might want to come back to this section to understand the subtleties of these classes.

The Basic Wrapper Classes

Java includes basic wrapper classes called `Character`, `Double`, and `Integer`. Each class wraps its basic type in an object, which stores the actual data and contains a variety of useful methods for this data type. To declare and optionally initialize a variable of a wrapper class, use:

```
WrapperClass varName [ = new WrapperClass(basicType) ];
```

where `WrapperClass` is either `Character`, `Double`, or `Integer`, and `basicType` is, correspondingly, a `char`, `double`, or `int` expression. The methods of the wrapper classes can be categorized into two groups:

- *methods tied to a variable of a wrapper type – to access these, use:*

varName.methodName(parameterList); methods tied directly to the wrapper class³¹ – to access these, use:

```
WrapperClass.methodName(parameterList);
```

Tables 2.39, 2.40, and 2.41 show some of the methods available for the basic wrapper classes³².

Method	Description
Methods bound to variables of type <code>Character</code>	
<code>char charValue()</code>	returns the <code>char</code> value of this <code>Character</code> object
<code>boolean equals(Object obj)</code>	returns <code>true</code> if the input represents the same character as the variable owning the operation
<code>String toString()</code>	returns a <code>String</code> representing this character
Methods bound to the class <code>Character</code>	
<code>boolean isLowerCase(char ch)</code>	returns <code>true</code> if input is a lowercase character
<code>boolean isUpperCase(char ch)</code>	returns <code>true</code> if input is an uppercase character
<code>boolean isDigit(char ch)</code>	returns <code>true</code> if the input character is a digit
<code>boolean isLetter(char ch)</code>	returns <code>true</code> if the input character is a letter
<code>char toLowerCase(char ch)</code>	returns lowercase version of the input character
<code>char toUpperCase(char ch)</code>	returns uppercase version of the input character

Table 2.39: Operations bound to `Character` variables and class

Example 2.27: Using Character operations

Create a method that takes as input a `String` and returns as output a `String` that contains the same characters as the input `String` but with capitals and lowers reversed.

Obtain the input string from the user via the `Console` class described in section 2.4.

Problem Analysis: We will use the methods `isLowerCase`, `isUpperCase`, `toLowerCase`, `toUpperCase`, and `isLetter`, all bound to the *class* `Character`. That means that we use the name of the class instead of a variable name to access the methods³³.

³¹ In Chapter 3 we will see that methods bound to a class are `static` methods.

³² The wrapper classes contain more methods than listed here, and there are additional wrapper classes not mentioned. Consult the Java API documentation for details, looking for the `java.lang` package.

³³ Recall that we can use the operation `length` of a `String` variable to get its length, and `charAt(i)` to extract its *i*-th character. Both methods are bound to `String` variables, not the `String` class.

1. We create a method `switchCase` that takes as input a `String` and returns the input string with all characters converted to their opposite case.
2. The standard `main` method uses the `readString` method of the `Console` class to obtain an input string from the user. The `Console` class must be located in the same directory as our current class. The method `switchCase` will then convert the input string.

Step 1: The method `switchCase` loops through all characters of the input string, checks if they are letters, and if so converts them to their opposite case. Then they are appended to a second string:

```
public static String switchCase(String s)
{
    String newString = new String("");
    for (int i = 0; i < s.length(); i++)
    {
        char cc = s.charAt(i);
        if (Character.isLetter(cc))
        {
            if (Character.isUpperCase(cc))
                cc = Character.toLowerCase(cc);
            else if (Character.isLowerCase(cc))
                cc = Character.toUpperCase(cc);
        }
        newString += cc;
    }
    return newString;
}
```

Step 2: If the `Console.java` file is located in the same directory as the following program, we can use `readString` as described in example 2.24, to finish our program:

```
public class ConvertString
{
    public static String switchCase(String s)
    {
        /* as above */
    }
    public static void main(String args[])
    {
        System.out.print("Enter input string: ");
        String input = Console.readString();
        System.out.println("String converted to: " + switchCase(input));
    }
}
```

Method	Description
Methods bound to variables of type <code>Double</code>	
<code>boolean isNaN()</code>	returns true if variable equals "Not-A-Number" NaN
<code>boolean isInfinite()</code>	returns true if variable equals plus or minus infinity
<code>double doubleValue()</code>	returns the double value of the variable
<code>boolean equals(Object obj)</code>	returns true if input variable is a <code>Double</code> with equal value as the variable owning this operation
<code>String toString()</code>	returns a <code>String</code> representing this value
Methods bound to the class <code>Double</code>	
<code>Double valueOf(String s)</code> throws <code>NumberFormatException</code>	converts the input string to <code>Double</code> , if possible, otherwise throws a <code>NumberFormatException</code>
<code>String toString(double d)</code>	returns a <code>String</code> representing the input value

Table 2.40: Operations bound to `Double` variables and class

Example 2.28: Using Double operations

Write some code that checks two double numbers `x` and `y` for equality. The code should take into account that doubles can be infinite or NaN. In each situation, the code should display exactly why the numbers are equal (if they are).

Problem Analysis: Usually the double-equal operator checks two doubles for equality but consider the following example:

```
double x = (4.0 - 4.0) / (6.0 - 6.0);
double y = 0.0 / 0.0;
System.out.println("x == y is " + (x == y));
```

That code segment will compile and execute without problems, but `x` and `y` will evaluate to "NaN" (not a number). That is mathematically correct, because you can not divide by zero. Even though both values are "NaN", the code will state that `x` and `y` are *not* equal.

Program Implementation: We have to take several cases into account: are the numbers "plain" numbers, NaN, or plus or minus infinity? Therefore the operations `isNaN` and `isInfinite` of the `Double` class will be useful, which are bound to variables. Assuming that `x` and `y` are the double values to check:

```
Double X = new Double(x);
Double Y = new Double(y);

if ((X.isNaN()) && (Y.isNaN()))
    System.out.println("Both doubles are NaN (not-a-number)");
else if ((X.isInfinite()) && (Y.isInfinite()) && (x * y > 0))
    System.out.println("Both doubles are infinite with the same sign");
else if (x == y)
    System.out.println("Both numbers are the same");
else
    System.out.println("The doubles are different");
```

Method	Description
Methods bound to variables of type <code>Integer</code>	
<code>int intValue()</code>	returns the <code>int</code> value of the variable
<code>boolean equals(Object obj)</code>	returns <code>true</code> if input variable is a <code>Double</code> with equal value as the variable owning this operation
<code>String toString()</code>	returns a <code>String</code> representation of this value
Methods bound to the class <code>Integer</code>	
<code>Integer valueOf(String s)</code> throws <code>NumberFormatException</code>	converts input to corresponding <code>Integer</code> , if possible, otherwise throws a <code>NumberFormatException</code>
<code>String toString(int i)</code>	returns a <code>String</code> representation of the input value

Table 2.41: Operations bound to `Integer` variables

Example 2.29: Using `Integer` operations

Write a method that converts a `String` entered on the command line to the corresponding `int` value. What happens if the string does not represent an integer?

The method `valueOf` bound to the class `Integer` seems appropriate, but it returns an `Integer`, not an `int`.³⁴ The method `intValue` returns an `int` but it is bound to a variable of type `Integer`. We will use both methods, first `valueOf` to create an `Integer` variable, then `intValue` applied to that variable:

```
public class IntegerConverter
{
    public static int convertToInt(String s)
    {
        Integer I = Integer.valueOf(s);
```

³⁴ We will ignore the keywords `throws NumberFormatException` for the `valueOf` method and explain it in chapter 5.

```

        return I.intValue();
    }
    public static void main(String args[])
    {
        if (args.length > 0)
            System.out.println(convertToInt(args[0]));
        else
            System.out.println("Nothing to convert");
    }
}

```

Figure 4 shows some tests of our conversion method:

```

C:\>java IntegerConverter
Nothing to convert
C:\>java IntegerConverter "123"
123
C:\>java IntegerConverter "Java"
Exception in thread "main" java.lang.NumberFormatException: Java
    at java.lang.Integer.parseInt(Integer.java:409)
    at java.lang.Integer.valueOf(Integer.java:511)
    at IntegerConverter.convertToInt(IntegerConverter.java:3)
    at IntegerConverter.main(IntegerConverter.java:8)

```

Figure 2.42: Testing the `convertToInt` method

When the input string does not represent an integer our program generates a runtime error. At this point there is little we can do about that, but chapter 5 will explain how to use Java's error-handling mechanism to improve such conversion methods. ■

Case Study: Solving a System of Linear Equations

In this optional section we create a program to solve a system of linear equations illustrating the usefulness of two-dimensional arrays. The program pulls together almost everything we have learned so far into one project. The example uses a fair amount of mathematics and you might want to take a brief look at a "Linear Algebra" text to recall how to solve a system of linear equations.

Example 2.30: Solving a system of linear equations

Create a program that can solve a system of N linear equations in N variables.

This is a complex task, but with the help of modularization (subtasks) we should be able to handle it just fine.

Problem Analysis: First, let's review the mathematics behind the problem so that we have a clear understanding of the steps involved. Here is an example of a system of three linear equations in three unknowns:

$$\begin{aligned}
 2x + 4y + 6z &= 10 \\
 x - 2y - 3z &= 6 \\
 2x - 2y + 4z &= 8
 \end{aligned}$$

To solve this system in an efficient and organized manner, we label the columns of a table with the names of the variables and add a separate column for the right-hand side. Then we fill in the numbers from the above equations, which gives us table 2.43:

x	y	z	RHS
2	4	6	10
1	-2	-3	6
2	-2	4	8

Table 2.43: Augmented coefficient matrix with headings

If an equation does not contain a variable, we enter a zero in the corresponding cell. From this table we extract the numbers only to form the two-dimensional array called the *augmented coefficient matrix*:

$$\begin{pmatrix} 2 & 4 & 6 & 10 \\ 1 & -2 & -3 & 6 \\ 2 & -2 & 4 & 8 \end{pmatrix}$$

The goal is to convert this matrix into another one called the normal form of the coefficient matrix that has the numbers 1 in the main diagonal and 0 everywhere else (except possibly the last column).³⁵ To achieve this, we can use the following operations:

Legal Matrix Operations:

- (A) You can swap any two rows.
- (B) You can multiply or divide an entire row by any number except zero.
- (C) You can replace any row by the sum of itself and a non-zero multiple of another row.

1. Make sure the number in position [0][0] is 1 by dividing the first row by 2 (operation B).

$$\begin{pmatrix} [2] & 4 & 6 & 10 \\ 1 & -2 & -3 & 6 \\ 2 & -2 & 4 & 8 \end{pmatrix} \Rightarrow \begin{pmatrix} [1] & 2 & 3 & 5 \\ 1 & -2 & -3 & 6 \\ 2 & -2 & 4 & 8 \end{pmatrix}$$

2. We use the 1st row to turn the first numbers in the 2nd and 3rd row to 0 by adding to them suitable multiples of the 1st row (operation C).

$$\begin{pmatrix} [1] & 2 & 3 & 5 \\ (1) & -2 & -3 & 6 \\ (2) & -2 & 4 & 8 \end{pmatrix} \Rightarrow \begin{pmatrix} [1] & 2 & 3 & 5 \\ (0) & -4 & -6 & 1 \\ 2 & -2 & 4 & 8 \end{pmatrix} \Rightarrow \begin{pmatrix} [1] & 2 & 3 & 5 \\ 0 & -4 & -6 & 1 \\ (0) & -6 & -2 & -2 \end{pmatrix}$$

3. We turn the number in position [1][1] to 1 by dividing the 2nd row by -4 (operation B).

$$\begin{pmatrix} 1 & 2 & 3 & 5 \\ 0 & [-4] & -6 & 1 \\ 0 & -6 & -2 & -2 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 3 & 5 \\ 0 & [1] & 1.5 & -0.25 \\ 0 & -6 & -2 & -2 \end{pmatrix}$$

4. We use the 2nd row to turn the second numbers in the 1st and 3rd row to 0 by adding to them suitable multiples of the 2nd row (operation C).

³⁵ Not every system of linear equations has a solution, and sometimes there is more than one solution. The normal form of the augmented matrix will tell us whether there is no solution, one solution, or more than one solution.

$$\begin{pmatrix} 1 & (2) & 3 & 5 \\ 0 & [1] & 1.5 & -0.25 \\ 0 & (-6) & -2 & -2 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & (0) & 0 & 5.5 \\ 0 & [1] & 1.5 & -0.25 \\ 0 & -6 & -2 & -2 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 & 5.5 \\ 0 & [1] & 1.5 & -0.25 \\ 0 & (0) & 7 & -3.5 \end{pmatrix}$$

5. We turn the number in position [2][2] to 1 by dividing the 3rd row by 7 (operation B)

$$\begin{pmatrix} 1 & 0 & 0 & 5.5 \\ 0 & 1 & 1.5 & -0.25 \\ 0 & 0 & [7] & -3.5 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 & 5.5 \\ 0 & 1 & 1.5 & -0.25 \\ 0 & 0 & [1] & -0.5 \end{pmatrix}$$

6. We use the 3rd row to turn the third numbers in the 1st and 2nd row to 0 by adding to them suitable multiples of the 3rd row (the 1st row is already in proper form) (operation C).

$$\begin{pmatrix} 1 & 0 & (0) & 5.5 \\ 0 & 1 & (1.5) & -0.25 \\ 0 & 0 & [1] & -0.5 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 & 5.5 \\ 0 & 1 & (0) & -0.5 \\ 0 & 0 & [1] & -0.5 \end{pmatrix}$$

We are done, because the last matrix is now in normal form. Converted back into a table it reads:

x	y	z	RHS
1	0	0	5.5
0	1	0	0.5
0	0	1	-0.5

Table 2.44: Normal form of augmented coefficient matrix with headings

Therefore $x = 5.5$, $y = 0.5$, and $z = -0.5$. A quick check plugging these values into the original equations will show that these answers are correct.

Program Implementation: Now we are ready to create a program that can do these calculations for us, and in fact can handle larger systems of equations just as well. We clearly need to be able to do the "valid matrix operations", so those are good candidates for separate methods:

- **Operation A:** Swap two rows, where the input indicates which rows to swap:

```
public static void swapRows(double matrix[][], int row1, int row2)
{
    for (int col = 0; col < matrix[row1].length; col++)
    {
        double tmp = matrix[row1][col];
        matrix[row1][col] = matrix[row2][col];
        matrix[row2][col] = tmp;
    }
}
```

- **Operation B:** Multiply a row by a number where the input indicates which row and number to use:

```
public static void multRow(double matrix[][], double number, int row)
{
    for (int col = 0; col < matrix[row].length; col++)
        matrix[row][col] *= number;
}
```

- **Operation C:** Replace a row by the sum of itself and a multiple of another row, where the input indicates which rows to add, and by what number to multiply one of them:

```
public static void addRows(double matrix[][], double num, int row1, int row2)
{
    for (int col = 0; col < matrix[row1].length; col++)
```

```

        matrix[row2][col] += num * matrix[row1][col];
    }

```

The input parameter `row2` also indicates that row `row2` is to be replaced while row `row1` will be multiplied by the number `num`.

In addition we will need methods to create the augmented coefficient matrix corresponding to the system to solve and to display a matrix. In example 2.17 we already created a `showMatrix` method, which we can reuse without change for our current program. A complete program to solve the above system of three equations in three unknowns could now look as follows:

```

public class EquationSolver
{
    public static void swapRows(double matrix[][],int row1,int row2)
    { /* as above */ }
    public static void multRow(double matrix[][],double number,int row)
    { /* as above */ }
    public static void addRows(double matrix[][],double num,int row1,int row2)
    { /* as above */ }
    public static void showMatrix(double matrix[][])
    { /* as in example 2.17 */ }
    public static double[][] createMatrix()
    {
        double matrix[][] = { {2,4,6,10},
                               {1,-2,-3,6},
                               {2,-2,4,8} };

        return matrix;
    }
    public static void main(String args[])
    {
        double matrix[][] = createMatrix();
        showMatrix(matrix);
        multRow(matrix, 1.0 / matrix[0][0], 0);
        addRows(matrix, -matrix[1][0], 0, 1);
        addRows(matrix, -matrix[2][0], 0, 2);

        multRow(matrix, 1.0 / matrix[1][1], 1);
        addRows(matrix, -matrix[0][1], 1, 0);
        addRows(matrix, -matrix[2][1], 1, 2);

        multRow(matrix, 1.0 / matrix[2][2], 2);
        addRows(matrix, -matrix[0][2], 2, 0);
        addRows(matrix, -matrix[1][2], 2, 1);

        showMatrix(matrix);
    }
}

```

This program solves at best a system of three equations whereas our goal is to create a program that can handle an arbitrary system. We therefore have to introduce additional methods

- A method `normalizeColumn` to manipulate a column until it contains only one 1 and the rest 0's:

```

public static void normalizeColumn(double matrix[][], int col)
{
    multRow(matrix, 1.0 / matrix[col][col], col);
    for (int row = 0; row < matrix.length; row++)
        if (row != col)
            addRows(matrix, -matrix[row][col], col, row);
}

```

For example, a call to `normalizeColumn(matrix, 2)` changes the entry in `[2][2]` to 1 and all other numbers in column 2 to 0.

- A method `normalizeMatrix` to apply `normalizeColumn` to all columns.

```
public static void normalizeMatrix(double matrix[][])
{   for (int col = 0; col < matrix.length; col++)
    normalizeColumn(matrix, col);
}
```

This method will apply `normalizeColumn` to all columns, up the number of rows of the matrix, because an augmented coefficient matrix has one more column than rows and we do not want to normalize the last column.

Now a single call to `normalizeMatrix` in the main method of our program will place 1's in the main diagonal and 0's elsewhere, and we can read off the solution by calling `showMatrix`:

```
public static void main(String args[])
{   double matrix[][] = createMatrix();
    normalizeMatrix(matrix);
    showMatrix(matrix);
}
```

Our program should now be able to solve any system of linear equations, but we made *one significant error*:

In the `normalizeColumn` method we divide the rows by their entry in the main diagonal but if that entry was zero our division – and the entire program – would fail.

Our solution for this dilemma is to try to swap rows until an entry in the main diagonal is not zero before calling `normalizeColumn`. But, as it turns out, that is not always possible. There are systems of equations that do not have a unique solution, or no solution at all, and this situation corresponds to the fact of not being able to find a non-zero entry in a particular column.³⁶ For our code that means that we need to add another method to swap columns if possible, or to inform us when this is not possible so that we know there is no unique solution.

- The method `foundNonZeroDiag` will return `true` and swap the rows if it finds a non-zero entry in the desired column, or return `false` and do nothing otherwise.

```
public static boolean foundNonZeroDiag(double matrix[][], int diag)
{   for (int row = diag; row < matrix.length; row++)
    {   if (matrix[row][diag] != 0)
        {   swapRows(matrix, row, diag);
            return true;
        }
    }
    return false;
}
```

For example, `foundNonZeroDiag(matrix, 2)` checks each row from 2 to the number of rows for a non-zero entry in column 2. If it finds one, it swaps rows and returns `true` so that the entry in `[2][2]` is now non-zero. If row 2 already had a non-zero entry in column 2, the method returns immediately. If no row 2 or below had a non-zero element in column 2, the method returns `false`.

Now, of course, we have to make sure that `normalizeMatrix` uses `foundNonZeroDiag`:

³⁶ For details about the mathematics, please refer to a book on linear algebra.

```

public static boolean normalizeMatrix(double matrix[][])
{ for (int col = 0; col < matrix.length; col++)
  { if (foundNonZeroDiag(matrix, col))
    normalizeColumn(matrix, col);
    else
      return false;
  }
  return true;
}

```

The new version returns `true` if it was able to normalize the matrix, or `false` if it could not do so based on the information provided by `foundNonZeroDiag`.

The final step is to change `createMatrix` so that the user can enter an arbitrary augmented coefficient matrix corresponding to the system to solve. We will use the `readDouble` and `readInt` methods of the `Console` class (see example 2.24) to obtain the user input (`Console.java` must be located in the same directory as `EquationSolver`). Here is the final version of our program:

```

public class EquationSolver
{ public static void swapRows(double matrix[][],int row1,int row2)
  { /* as above */ }
  public static void multRow(double matrix[][],double number,int row)
  { /* as above */ }
  public static void addRows(double matrix[][],double num,int row1,int row2)
  { /* as above */ }
  public static boolean foundNonZeroDiag(double matrix[][], int diag)
  { /* as above */ }
  public static void normalizeColumn(double matrix[][], int col)
  { /* as above */ }
  public static boolean normalizeMatrix(double matrix[][])
  { /* as above (modified version) */ }
  public static void showMatrix(double matrix[][])
  { /* as in example 2.17 */ }
  public static double[][] createMatrix()
  { System.out.print("Enter number of variables: ");
    int numVars = Console.readInt();
    double matrix[][] = new double[numVars][numVars+1];
    for (int row = 0; row < numVars; row++)
    { for (int col = 0; col < numVars; col++)
      { System.out.print("Coefficient row " + (row+1) +
        ", column " + (col+1) + ": ");
        matrix[row][col] = Console.readDouble();
      }
      System.out.print("Right-hand side for row " + (row+1) + ": ");
      matrix[row][numVars] = Console.readDouble();
    }
    return matrix;
  }
  public static void main(String args[])
  { double matrix[][] = createMatrix();
    showMatrix(matrix);
    if (normalizeMatrix(matrix))
      showMatrix(matrix);
    else
      System.out.println("No solution or no unique solution");
  }
}

```

We have now a working program that can solve any system of linear equations that has a unique solution. The next step is to test of our program.

Testing the Program: Figures 2.45, 2.46, and 2.47 show three runs of our equation solver.

Here is our original system of equations:

$$\begin{aligned} 2x + 4y + 6z &= 10 \\ x - 2y - 3z &= 6 \\ 2x - 2y + 4z &= 8 \end{aligned}$$

As we previously worked out "by hand", the solution is $x = 5.5$, $y = 0.5$, and $z = -0.5$, just as shown by our program.

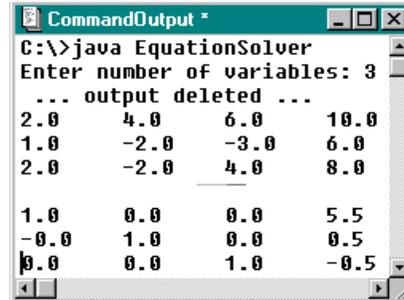


Figure 2.45: Solved system of equation

Here is a system that has no solution:

$$\begin{aligned} z &= 1 \\ y &= 1 \\ 2y &= 1 \end{aligned}$$

Our program confirms that, so we are looking good.

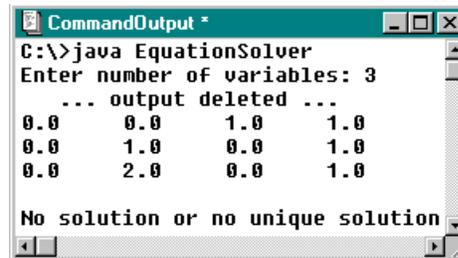


Figure 2.46: System of equations with no solution

Finally, here is a (randomly chosen) system of six equations with six unknowns:

$$\begin{aligned} 2x_1 + 4x_2 + 6x_3 + x_4 + x_5 + x_6 &= 2 \\ x_1 + 2x_2 + 3x_3 + 6x_4 + 6x_5 + 5x_6 &= 4 \\ 2x_1 + 2x_2 + 4x_3 + 8x_4 + 2x_5 + 9x_6 &= 7 \\ & & & & x_5 & & = 1 \\ 2x_1 + & & 3x_3 + & & 4x_5 & & = 5 \\ & x_2 + & & 2x_4 + & & 3x_6 & = 2 \end{aligned}$$

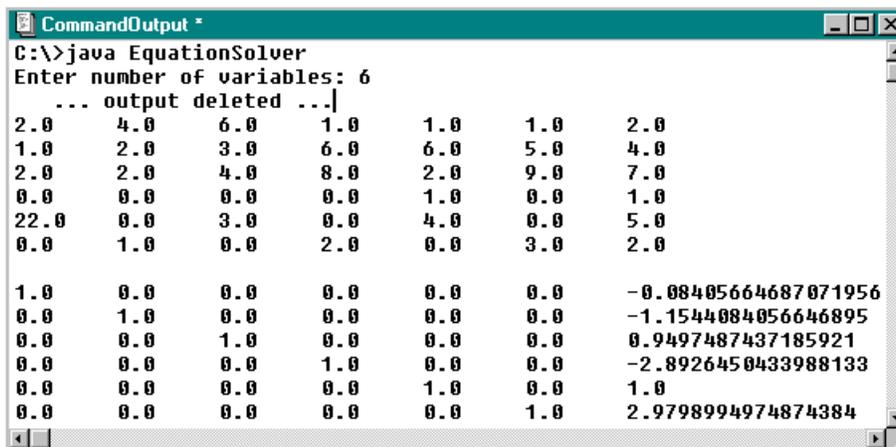


Figure 2.47: Solved system of six equations with six unknown variables

Using a calculator (and a little bit of time and patience) we can confirm that our program again delivered the correct solution. We therefore pronounce it as working properly.

Program documentation: The last step is to create proper documentation for our methods. We will of course use documentation comments and the javadoc tool. Our commented class looks as follows:

```

/** Solves a system of N linear equations in N variables. It asks the user to
    enter the augmented coefficient matrix corresponding to the system of
    equations to solve. The program will deliver the normalized coefficient
    matrix if there is a unique solution, or it will inform the user that there
    is no solution, or no unique solution. Requires class <b>Console</b>.
    @author Bert G. Wachsmuth
    @version 1.0 (12/08/99)
*/
public class EquationSolver
{ /** Swap row <code>row1</code> and <code>row2</code> in matrix
   <code>matrix</code> */
  public static void swapRows(double matrix[][],int row1,int row2)
  /** Multiplies row <code>row</code> of matrix <code>matrix</code> with
    <code>number</code>. The answer replaces row <code>row</code>. */
  public static void multRow(double matrix[][],double number,int row)
  /** Multiplies row <code>row1</code> by <code>num</code>, adds the result
    to <code>row2</code>, and replaces row <code>row2</code>. */
  public static void addRows(double matrix[][],double num,int row1,int row2)
  /** Finds non-zero entry at position <code>[diag][diag]</code>. If possible,
    swaps rows so that row <code>diag</code> will have non-zero entry in
    column <code>diag</code> and returns true. Otherwise returns false. */
  public static boolean foundNonZeroDiag(double matrix[][], int diag)
  /** Changes all entries in column <code>col</code> so that the entry at
    position <code>[col][col]</code> is 1, all others in that column 0.
    <b>The method assumes that matrix[diag][diag] is not zero.</b> */
  public static void normalizeColumn(double matrix[][], int col)
  /** Converts matrix <code>matrix</code> into normal form is possible and
    returns true. Otherwise returns false. */
  public static boolean normalizeMatrix(double matrix[][])
  /** Displays matrix <code>matrix</code> on standard output. */
  public static void showMatrix(double matrix[][])
  /** Asks the user to enter matrix. User input is obtained via the Console
    class, which must be present in the same directory as this class. */
  public static double[][] createMatrix()
}

```

Chapter Summary

In this chapter we introduced the following concepts and terminology:

Methods, Java Program, Method Exit Point

See section 2.1, examples 2.01 (*Using methods to compute properties of a square*) and 2.02 (*Using methods to find the sum of prime numbers*)

Input Parameters for Methods

See section 2.1, examples 2.03 (*Illustrating value parameters*) and 2.04 (*Illustrating valid input expressions*)

Reference Type, Checking Reference Variables for Equality

See section 2.1, example 2.05 (*Equality of String types*)

Reference Types as Input Parameters

See section 2.1, example 2.06 (*Using reference types as method input parameters*)

Fields and Local Variables

See section 2.2, example 2.07 (*Illustrating fields and local variables*)

Scope Rules

See section 2.2, examples 2.08 (*Scope of fields and local variables*), 2.09 (*Compound interest computation using fields*), and 2.10 (*Illustrating the scope rule*)

Declaring Arrays, Initializing Arrays

See section 2.3, example 2.11 (*System.out.println applied to arrays*)

Accessing Array Elements

See section 2.3, examples 2.12 (*Displaying an array*) and 2.13 (*Arrays of String*)

Arrays as Method Parameters

See section 2.3, examples 2.14 (*Arrays as method input parameters*) and 2.15 (*Compound interest computation using arrays*)

Command Line Input

See section 2.3, example 2.16 (*Command line input to main method*)

Two-Dimensional Arrays

See section 2.3, examples 2.17 (*Working with two-dimensional arrays*), 2.18 (*Two-dimensional array representing a table of sales figures*), and 2.19 (*Triangular array for prefix sales averages*)

Implementation Comments

See section 2.4, examples 2.20 (*Proper and improper use of comments*) and 2.21 (*Proper use of comments*)

Documentation Comments

See section 2.4, example 2.22 (*Using javadoc comments*)

The javadoc Tool

See section 2.4, example 2.23 (*Using the javadoc tool*)

The Console Class

See section 2.4, examples 2.24 (*The Console class for simplified user input*) and 2.25 (*Computing Exam Average and Standard Deviation*)

Error Types

See section 2.4, example 2.26 (*Error types and debugging tips*)

The Basic Wrapper Classes

See section 2.5, examples 2.27 (*Using Character operations*), 2.28 (*Using Double operations*), and 2.29 (*Using Integer operations*)

Case Study: Solving a System of Linear Equations

