Programming Fundamentals

Hyesung Park, Na’el Abu-halaweh, Sonal S. Dekhane, Wei Jin, Robert Lutz, Richard W. Price, Tacksoo Im

Georgia Gwinnett
COLLEGE

Table of Contents

1. Getting Started 9
1.1. Learning Outcomes 9
1.2. Key Terms 9
1.3. Resources 9
1.3.1. Text 9
1.3.2. Video/Tutorial 9
1.4. Overview 10
1.5. Basic Computer Architecture 11
1.6. Programming Basics 11
1.6.1. What is Computer Science? 11
1.6.2. What is a Computer? 11
1.6.3. Early Computing 11
1.6.4. What is Programming? 11
1.7. Binary Number System (Resources) 12
1.7.1. Complements 12
1.7.2. Decimal Complements 13
1.7.3. Binary Complements 13
1.8. Algorithms 14

1.8.1. Algorithms are Everywhere! 14
1.8.2. What are Algorithms? 14
1.8.3. Algorithm Examples 15
1.9. Code Conventions for the Java Programming Language 16
1.10. What is Java 16
1.11. History of Java 17
1.11.1. History of Java by javapoint 17
1.11.2. A Short History of Java 17
1.12. Java Versions 17
1.13. Features of Java Programs 18
1.13.1. Basic Form of the first program "MyFirstJava" 18
1.13.2. JavaTutor Example 19
1.13.3. Use of print vs. println MyPrintln 19
1.13.4. JavaTutor Example 19
1.13.5. Use of println with ThreeMessages 20
1.13.6. JavaTutor Example 20
1.13.7. Simple Computation 20
1.13.8. JavaTutor Example 20
1.13.9. Escape Sequences 20
1.13.10. Programming Errors in General (not only for Java) 20
1.14. Creating, Compiling, and Executing a Java Program 21
1.14.1. Video: JDK 12 install in Windows 10 21
1.14.2. Video: JDK 12 install on Mac OS X 21
1.14.3. Compiling and Running a Simple Program 21
1.15. Exercises 21
1.15.1. Write a program named Banana.java that displays as below: 21
1.15.2. Write a program named Fibonacci.java that displays the result of 21
1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 21
1.15.3. Determine the nines complements of the decimal numbers: 21
1.15.4. Determine the one’s and two’s complements of the binary numbers: 21

1.15.5. Convert each binary number to its decimal equivalent: 21
1.15.6. Convert each decimal number to its binary equivalent: 22
1.15.7. Write a program named Formula.java that displays the result of........ 22
$5.6 \times 5.6 - 4 \times 6.2 \times 5.12 \times 7.8 - 3 \times 5.6$.. 22
1.15.8. Write a program named Molecular.java that displays the following table:.. 22
1.15.9. SpeedLight.java 22
1.15.10. Chocolate.java... 22
1.15.11. Stamps.java 23
1.15.12. Cycle.java 23
1.15.13. FindX.java.. 23
1.15.14. MaleStudent.java 23
1.15.15. Circle.java 23
1.16. Do You Have Any Questions about Chapter 1? 23
2. Datatype, Variables, and Expressions.. 24
2.1. Learning Outcomes.. 24
2.2. Key Terms... 24
2.3. Resources.. 24
2.4. Overview.. 24
2.5. Variable... 26
2.6. Data Type... 26
2.7. Declaration Statement: Declare Data Type for a Variable 27
2.8. Assignment Statement: Change the Value of a Variable.................................. 29
2.8.1. JavaTutor Example 30
2.9. Expressions 30
2.10. Output Statement... 31
2.10.1. Formatting outputs using printf.. 32
2.11. Comments and Order of Execution.. 32
2.12. Declare Constants and Get User Input with Scanner 33
2.12.1. Create a Scanner Object.. 36

2.12.2. Invoke Methods on the Scanner Object to Retrieve Information 38
2.13. Augmented Assignment Operators 39
2.14. Exercises 40
2.15. Do You Have Any Questions about Chapter 2? 44
3. 3. Conditions 44
3.1. Learning Outcomes 44
3.2. Key Terms 44
3.3. Resources 44
3.4. Overview 45
3.5. Relational Operators, Simple Boolean Expressions and the boolean Data type: 45
3.6. Logical Operators and Compound Boolean Expressions 47
3.7. Operator Precedence 49
3.8. if Statement 49
3.8.1. if Statement 49
3.8.2. if-else Statement 52
3.8.3. Multi-branch (Cascaded) if Statement 54
3.8.4. Nested if Statement 56
3.9. switch Statement 57
3.10. Summary 60
3.11. Exercises/Problem Solving: 60
3.12. Do You Have Any Questions about Chapter 3? 62
4. Chapter 4: Loops 62
4.1. Learning Outcomes 62
4.2. Key Terms 62
4.3. Resources 62
4.3.1. Text 63
4.3.2. Videos 63
4.4. Introduction 63
4.5. For Loops 63

4.6. While Loops 63
4.7. Common Loop Algorithms 64
4.7.1. Java Tutor Example 64
4.7.2. Java Tutor Example 64
4.7.3. Java Tutor Example 65
4.7.4. Sequence generating loops 65
4.8. Do-while Loops 65
4.9. break and continue 66
4.10. Nested Loops 66
4.11. Exercises 66
4.11.1. Exercise 1 67
4.11.2. Exercise 2 67
4.11.3. Exercise 3 67
4.11.4. Exercise 4 67
4.11.5. Exercise 5 67
4.11.6. Exercise 6 68
4.11.7. Exercise 7 68
4.11.8. Exercise 8 68
4.11.9. Exercise 9 68
4.11.10. Exercise 10 69
4.12. Do You Have Any Questions about Chapter 4? 69
5. Methods 69
5.1. Learning Objectives 69
5.2. Resources 69
5.2.1. Text 69
5.3. Key Terms 70
5.3.1. Think Java Vocabulary Chapter 4 70
5.3.2. Think Java Vocabulary Chapter 6 70
5.4. Overview 71
5.5. Method Basics 71

5.6. Example Method ... 71
5.7. Java Provided Methods ... 72
5.8. Create Your Own Methods ... 72
5.8.1. Parts of a Method ... 72
5.8.2. Method Header ... 72
5.8.3. Parameter Types ... 73
5.8.4. Primitive Parameters ... 73
5.8.5. Reference Parameters ... 74
5.8.6. Method Signature ... 74
5.8.7. Parameters ... 74
5.8.8. Method Body ... 75
5.9. Types of Methods ... 75
5.9.1. Class methods ... 75
5.9.2. JavaTutor Example ... 75
5.9.3. Instance methods ... 75
5.9.4. JavaTutor Example ... 76
5.10. Overloaded Methods ... 76
5.10.1. JavaTutor Example ... 76
5.11. Calling Methods ... 76
5.12. Member variables ... 77
5.12.1. Declaring member variables ... 77
5.13. Exercises ... 79
5.14. Do You Have Any Questions about Chapter 5? ... 83
6. Arrays and ArrayLists ... 83
6.1. How Does This Topic Relate to Object Oriented Programming? ... 83
6.2. Learning Objectives ... 83
6.3. Key Terms ... 83
6.4. Resources ... 84
6.4.1. Text ... 84
6.4.2. Video / Tutorial ... 84

6.5. Overview 84
6.6. Arrays 84
6.6.1. Creating Arrays 84
6.6.2. JavaTutor Example 84
6.6.3. Indexing Arrays 84
6.6.4. JavaTutor Example 85
6.6.5. Array Properties 85
6.6.6. JavaTutor Example 85
6.6.7. Arrays and For Loops 85
6.6.8. JavaTutor Example 85
6.6.9. JavaTutor Example 85
6.6.10. Arrays Example 85
6.6.11. JavaTutor Example 85
6.7. ArrayLists 86
6.7.1. Creating ArrayLists 86
6.7.2. JavaTutor Example 86
6.7.3. Indexing ArrayLists 87
6.7.4. JavaTutor Example 87
6.7.5. ArrayList Maintenance - Adding and Removing Elements 87
6.7.6. JavaTutor Example 87
6.7.7. ArrayList Methods 87
6.7.8. JavaTutor Example 87
6.7.9. ArrayLists and For Loops 87
6.7.10. JavaTutor Example 87
6.7.11. Dice Rolling Example, Revisited 88
6.7.12. JavaTutor Example 88
6.7.13. ArrayList to Array Conversion, and Vice-Versa 88
6.7.14. JavaTutor Example 88
6.8. Exercises 88
6.8.1. Exercise 1 88

6.8.2. Exercise 2 88
6.8.3. Exercise 3 88
6.8.4. Exercise 4 89
6.8.5. Exercise 5 89
6.8.6. Exercise 6 89
6.8.7. Exercise 7 90
6.8.8. Exercise 8 90
6.9. Do You Have Any Questions about Chapter 6? 91
7. Object Oriented Programming 91
7.1. Learning Objectives 91
7.2. Resources 91
7.2.1. Text 91
7.2.2. Video 91
7.3. Key Terms 91
7.4. Overview 91
7.5. Classes and Objects 92
7.6. Defining a class 92
7.7. Constructor 93
7.8. Getters and Setters 94
7.9. Static vs. Instance (non-static) methods 95
7.10. Implementing the *equals()* and *toString()* method 95
7.11. Using two or more classes together 96
7.12. Exercises 97
7.12.1. Create the Student Class 97
7.12.2. Create the *equals()* and *toString()* method for the Student Class 97
7.12.3. Create the School Class 97
8. Do You Have Any Questions about Chapter 7? 98

1. Getting Started

1.1. Learning Outcomes

Students will be able to

- Understand computer basics.
- Understand programming basics.
- Understand binary number system.
- Begin using the Java programming language.
- Display output on the console.
- Explain the differences between syntax errors, runtime errors, and logic errors.

1.2. Key Terms

Review the important terms.

1.3. Resources

1.3.1. Text

- Think Java Computer Programming by Allen Downey and Chris Mayfield.
- Core Java : Core Java Complete by Cay S. Horstmann
- Essentials of the Java Programming: Essentials by Oracle.com
- Memory Bits and Bites
- Bits and Bytes
- Convert a decimal number to binary numbers
- How the Binary Number System Works
- Binary Addition
- Binary Subtraction
- Method of Complements

1.3.2. Video/Tutorial

- Core Java 11: Fundamentals by Cay S. Horstmann
- An Introduction to Java link
- Understand the Fundamental Concepts of Object-Oriented Programming Why OOP?
- Early Computing
- What is Algorithm?

- Kahn Academy: How Computers Work
- YouTube Video on Fetch Decode Execute Cycle

1.4. Overview

You might have heard about computers many times. You might have questions about how computers perform so many tasks. You might have questions about programming. What is programming? Welcome! If you start to ask all these questions, then you are in the right place to start to learn about programming. We use programming to develop software. A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial. It can be a symbolic computation, like searching and replacing text in a document or compiling a program.

Learning a programming language is similar to learning a foreign language, such as French. The ultimate goal of learning a foreign language is to be able to compose sentences to communicate (the communication problem). Similarly, the ultimate goal of learning to program is to write proper statements that solve a particular problem, such as calculating the volume of a sphere or calculating the projectile of a rocket.

A programming language offers its set of different types of statements for programmers to use. Before you can write code using a type of statements, you need to learn its

1. syntax
2. semantics

For French, syntax is the grammar of the language and semantics is what a sentence means. For programming, when we specify how the code should look like and structural rules, we are talking about syntax. Programming languages, such as Java, have very strict syntax rules, much stricter than a natural language (e.g. French). If your code does not comply with the syntax rules, the compiler (the translator) will not understand your code and therefore cannot translate your code into machine code that can be executed on CPU.

In order to learn how to compose the right sentences in French to convey certain meanings, you will normally first do some reading comprehension exercises to learn how to interpret what sentences mean. Similarly, in order to write code to solve a problem, you first need to read code written by others and understand what that code does. What a piece of code does is similar to the meaning of a sentence in a foreign language. We call this semantics of the code. Only if you understand the semantics (the behavior) of code can you compose the right code to have the desired behaviors.

Through syntax and semantics learning, you will build your tool set of a programming language's different types of statements. When the times come to write code to solve a problem, you need to pick up the right statements and use them in the right way and right order to achieve the desired behavior.

1.5. Basic Computer Architecture

The semantics of statement (an instruction written in high-level programming language, defined in the section above) is what it does on a computer. We need some very basic understanding and vocabulary in order to be able to state the behavior of a statement.

First, get some understanding of How Computer Works by watching the fun videos on Khan Academy: Khan Academy’s How Computer Works. Please watch the last two videos "CPU, Memory, Input and Output" and "Hardware and Software" (staring Bill Gates and other real technical people).

Second, watch this video on YouTube for a more detailed explanation about how CPU execute instructions in stored in memory: YouTube video (It’s less than 8 mins and please be patient and watch the whole video.)

We can see that memory holds instructions and data. Each memory location has a unique address. Instructions are fetched by CPU and executed by CPU. Besides the instructions, memory also holds data that need to be manipulated by the program.

1.6. Programming Basics

Computer programs, known as software, are instructions to the computer.

You tell a computer what to do through programs. Without programs, a computer is an empty machine. Computers do not understand human languages, so you need to use computer languages to communicate with them.

Programs are written using programming languages.

1.6.1. What is Computer Science?

1.6.2. What is a Computer?

1.6.3. Early Computing

1.6.4. What is Programming?

High-level languages:

Java, Python, C, C++, PHP, Ruby, and JavaScript.

Low-level languages:

It is a machine language and only a computer is capable of reading and interpreting the low-level languages of a collection of binary digits or bits.

How do low-level languages and high-level language work (Compilation of Java Programs)?

> *Before programs can run, programs in "high-level languages" have to be translated into a "low-level language", also called "machine language". This translation takes some time, which is a small disadvantage of high-level languages.*
>
> *But high-level languages have two major advantages:*
>
> *It is much easier to program in a "high-level language". Programs take less time to write, they are shorter and easier to read, and they are more likely to be correct. "High-level languages" are portable, meaning they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.*
>
> *Two kinds of programs translate "high-level languages" into "low-level languages: "interpreters and compilers". An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.*

(Read more from the Compiling Java Program)

1.7. Binary Number System (Resources)

- Memory Bits and Bytes
- Encoding Information
- Decimal to Binary
- Binary & data
- The Binary Number System
- Binary Addition
- Binary Subtraction

1.7.1. Complements

In mathematics and computing, there are two types of complements, the radix-minus-one complement and the radix complements. The complement by itself is the radix complement.

These complements are used to make the arithmetic operation in the digit system easier.

While we are familiar with decimal system, we will discuss the decimal complements first. In the decimal system, there are the nines complement and tens complement.

1.7.2. Decimal Complements

If A is a decimal number, the nines complement of A is obtained by subtracting A from (10^n – 1); and the tens complement of A is that we add 1 with the nines complement of A.

For example

	Example 1	*Example 2*	*Example 3*
Decimal number	4206	9674	124134
Nines complement	5793	0325	875865
Tens complement	5794	0326	875866

We can illustrate the use of complements by taking 4206. The nine's complement of 4206 will be

$$9999 - 4206 = 5793$$

Then, ten's complement of 5793 is 5794 as below.

$$5793 + 1 = 5794$$

1.7.3. Binary Complements

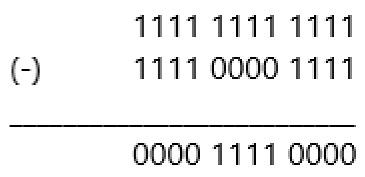
The principles of complements in the decimal system can be translated into the binary digit system. If B is a binary number, the one's complement of B is obtained by subtracting each digit of B from 1, and the two's complement of B is obtained by adding 1.

For example

		Number
1	Binary number	1111 0000 1111
2	One's complement	0000 1111 0000
3	Two's complement	0000 1111 0001

We can illustrate the example of binary complements as below.

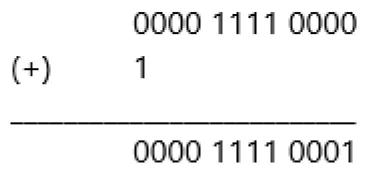
* Step 1: First, make the one's complement of 1111 0000 1111.



```
         1111 1111 1111
(-)      1111 0000 1111
_______________________
         0000 1111 0000
```

Simply you will change 0 to 1 and 1 to 0.

* Step 2: Two's complement of One's complement of 1111 0000 1111 which is now 0000 1111 0000. You are adding 1 to 0000 1111 0000 to make it Two's complement.



```
         0000 1111 0000
(+)      1
_______________________
         0000 1111 0001
```

More example resources

Two’s Complement
Ones' complement

1.8. Algorithms

1.8.1. Algorithms are Everywhere!

- Amazon.com
- Bank Systems
- Cash Registers
- Hospitals
- Internet Browsers
- Search Engines

1.8.2. What are Algorithms?

An algorithm specifies a series of steps that perform a particular computation or task. Algorithms were originally born as part of mathematics - the word "algorithm" comes from the writer Muhammad ibn Musa-al-Khwarizmi - but currently the word is associated with computer science.

More background information about Algorithms

In programming, each step should be clear and defined precisely to solve a particular problem most effectively.

What is Algorithm in brief? - Video Clip Resource

1.8.3. Algorithm Examples

Write an algorithm to add two numbers entered by user:

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum. sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

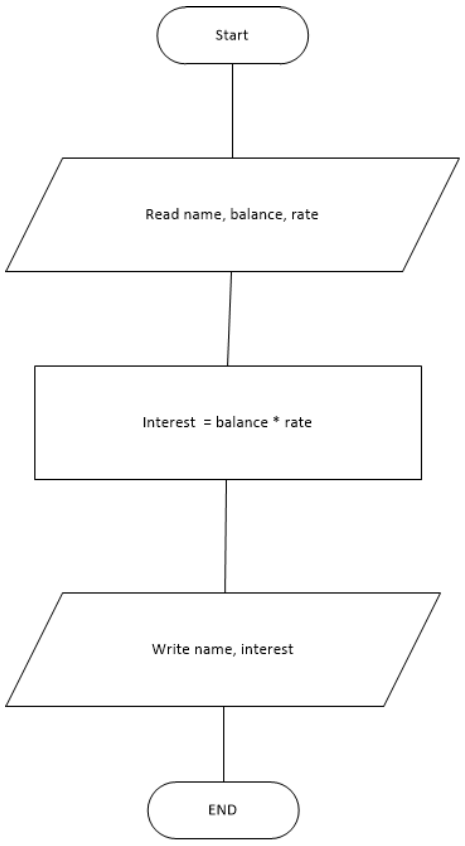
Write an algorithm to calculate and record the interest on a home mortgage for a client.

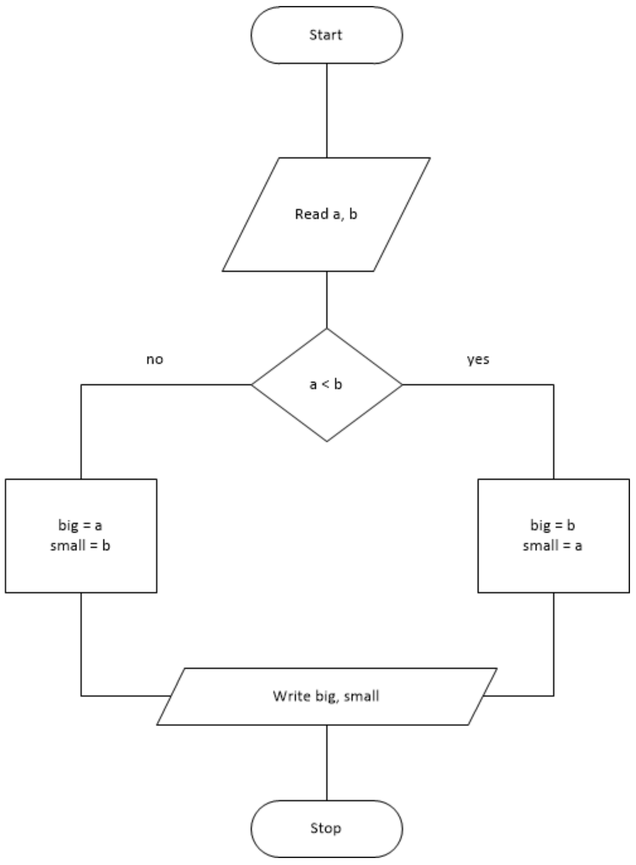
(Assuming that balance of the mortgage and the interest rate appear in the client's record - a collection of related data items. e.g. data on a given client, and a file is a collection of similar records) - Flowchart is provided in Figure 1.

```
Step 1: Start

Step 2: Obtain name, balance, and rate from the client's record.

Step 3: Compute: interest = balance * rate.

Step 4: Record client's name and interest in the interest file.
```

More examples of Algorithm in Programming

There are two ways of presenting an algorithm. One way is by a flowchart and another way is to write it in pseudocode language. Figure 1 shows two flowcharts. First one is based on the mortgage interest algorithm and second flowchart reads two numbers, a and b. Then prints them in descending order, after assigning the larger number to big and the smaller number to small. Two arrows leaving the decision if a < b. If the answer is no, it is labeled "no" and the other labeled "yes".





1.9. Code Conventions for the Java Programming Language

Coding Standards
Code Conventions by Oracle

1.10. What is Java

Java is one of the most widely used computer programming languages. According to Hostmann (2016), Java is a well-designed programming language and is an efficient and secure environment with a huge library.
The White Paper Buzzwords

- Simple
- Object Oriented
- Distributed
- Robust
- Secure
- Architecture-Neutral

- Portable
- Interpreted
- High-Performance
- Multithreaded
- Dynamic

1.11. History of Java

1.11.1. History of Java by javapoint

1.11.2. A Short History of Java

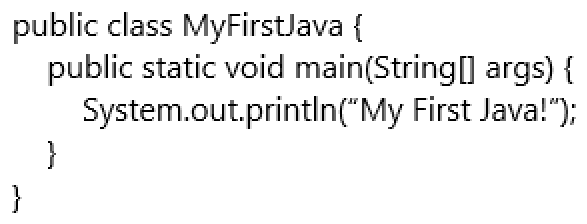
1.12. Java Versions

Version	Release Date	Extended Support Until
JDK Beta	1995	
JDK 1.0	January 1996	
JDK 1.1	February 1997	
J2SE 1.2	December 1998	
J2SE 1.3	May 2000	
J2SE 1.4	February 2002	February 2013
J2SE 5.0	September 2004	April 2015
Java SE 6	December 2006	December 2018
Java SE 7	July 2011	July 2022
Java SE 8	March 2014	March 2025
Java SE 9	September 2017	N/A
Java SE 10	March 2018	N/A
Java SE 11	September 2018	September 2026

Version	Release Date	Extended Support Until
java SE 12	March 2019	N/A
java SE 13	September 2019	N/A
java SE 14	March 2020	N/A

1.13. Features of Java Programs

1.13.1. Basic Form of the first program "MyFirstJava"



```
public class MyFirstJava {
   public static void main(String[] args) {
      System.out.println("My First Java!");
   }
}
```

When MyFirstJava program runs it displays:

My First Java!

If you compare this output with the original code of **MyFirstJava**, you will find there is no double quotations. My First Java! is a String literal in the code and it is defined by using double quotation like System.out.println("My First Java!"). System.out.println is to display the String literal you want to print out on the console/screen (Read more)

> *"System.out.println displays results on the screen; the name println stands for "print line". Confusingly, print can mean both "display on the screen" and "send to the printer". In this book, we'll try to say "display" when we mean output to the screen. Like most statements, the print statement ends with a semicolon (;).*
>
> *Java is "case-sensitive", which means that uppercase and lowercase are not the same. In this example, System has to begin with an uppercase letter; system and SYSTEM won't work."*

Following link shows how this MyFirstJava program works step by step by using the Java Visualizer program. MyFirstJava.java by Java Visualizer or use following embedded visualizer by scrolling down and to the right to expand the window.

Class and Methods in brief

A **method** is a named sequence of statements. This program defines one method named main:

public static void main(String[] args) The name and format of main is special: when the program runs, it starts at the first statement in main (All java programs begin here) and ends when it finishes the last statement. Later, we will see programs that define more than one method.

A **class** is a collection of methods. In this program defines a class named MyFirstJava. You can give a class any name you like, but it is conventional to start with a capital letter. The name of the class has to match the name of the file it is in, so this class has to be in a file named **MyFirstJava.java**.

Java uses **curly braces ({ and })** to group things together. In MyFirstJava.java, the outermost braces contain the class definition, and the inner braces contain the method definition.

1.13.2. JavaTutor Example

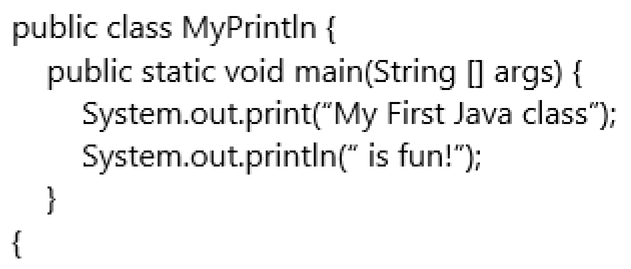
[JavaTutor Example] **(click "Vizualize Execution")**

System.out.println appends a newline. This newline starts from the beginning of the next line. If you want to make the String phrases appear in one line, then use **print** instead of **println** (see below example).

ClassName is written by a programmer and from the example above, MyFirstJava is the class name.

main is a special method which makes the program execute and **println** means display a message on the screen. The println statement terminates with a semi-colon(;).

1.13.3. Use of print vs. println MyPrintln

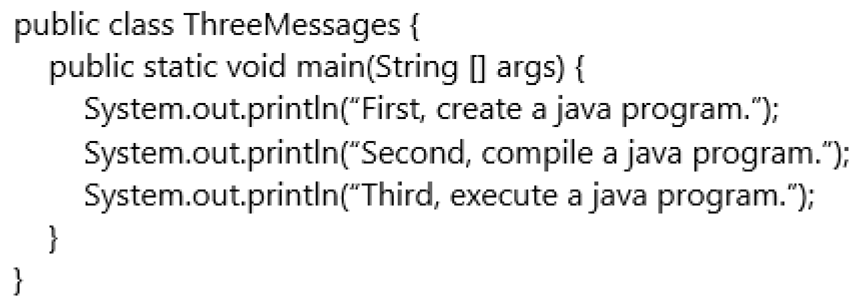


```
public class MyPrintln {
   public static void main(String [] args) {
      System.out.print("My First Java class");
      System.out.println(" is fun!");
   }
{
```

1.13.4. JavaTutor Example

[Java Tutor Example] **(click "Vizualize Execution")**

1.13.5. Use of println with ThreeMessages

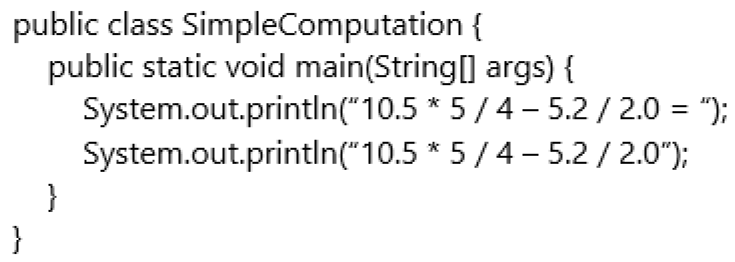


```
public class ThreeMessages {
   public static void main(String [] args) {
      System.out.println("First, create a java program.");
      System.out.println("Second, compile a java program.");
      System.out.println("Third, execute a java program.");
   }
}
```

1.13.6. JavaTutor Example

Java Tutor Example (click "Vizualize Execution")

1.13.7. Simple Computation



```
public class SimpleComputation {
   public static void main(String[] args) {
      System.out.println("10.5 * 5 / 4 – 5.2 / 2.0 = ");
      System.out.println("10.5 * 5 / 4 – 5.2 / 2.0");
   }
}
```

1.13.8. JavaTutor Example

Java Tutor Example (click "Vizualize Execution")

1.13.9. Escape Sequences

- How to use escape sequences
- From Java Tutorials

1.13.10. Programming Errors in General (not only for Java)

- Syntax errors
- Runtime errors
- Logic errors

1.14. Creating, Compiling, and Executing a Java Program

You must first set up the environment to compile and execute java programs. To be able to compile and run programs, you must install the JDK and configure it. Java SE 12.0.1 is the latest release for the Java SE Platform (as of June 09, 2019). Orale JDK

1.14.1. Video: JDK 12 install in Windows 10

1.14.2. Video: JDK 12 install on Mac OS X

1.14.3. Compiling and Running a Simple Program

1.15. Exercises

1.15.1. Write a program named Banana.java that displays as below:

A banana is an edible fruit.
If you wait the correct amount of time for it to ripen,
it will be sweet and delicious.

1.15.2. Write a program named Fibonacci.java that displays the result of

$$\mathbf{1 + 1 + 2 + 3 + 5 + 8 + 13 + 21}$$

1.15.3. Determine the nines complements of the decimal numbers:

4535
507606
78534019

1.15.4. Determine the one's and two's complements of the binary numbers:

110011
10001100
10111011101

1.15.5. Convert each binary number to its decimal equivalent:

101
110110
111011001
101010101

1.15.6. Convert each decimal number to its binary equivalent:

2857
4503
46098
694

1.15.7. Write a program named Formula.java that displays the result of

$$\frac{5.6 \times 5.6 - 4 \times 6.2 \times 5.1}{2 \times 7.8 - 3 \times 5.6}$$

1.15.8. Write a program named Molecular.java that displays the following table:

Atom	Weight (grams / molecule)
H	1.00794
C	12.0107
O	15.9994

1.15.9. SpeedLight.java

The speed of sound is approximately 340 meter per second. Assume that you just saw a lightning flash and you heard the sound of thunder 5 seconds later. Write a program named SpeedLight.java that calculate the distance to a lightning strike based on the time elapsed between the flash and the sound of thunder.

1.15.10. Chocolate.java

Assume there are 9 bags of chocolate bars. Each bag has two chocolate bars. The bag is big enough to have three chocolate bars. If you want to take all the chocolates out of each bag and add three chocolate bars to each bag, how many bags will you need? Write a program to compute the number of bags you will need to add three chocolates instead of two chocolates.

1.15.11. Stamps.java

Susan and Jean just started collecting stamps as a hobby. Susan has 8 endangered animal collection stamps and Jean has 40 racing car collection stamps. How many more does Jean have than Susan? Write a program named Stamps.java that compute the difference between Jean's and Susan's collections of the stamps.

1.15.12. Cycle.java

In the Cycle shop, there are 10 bicycles and X numbers of tricycles. Assume that you count the number of wheels of the bicycles and there are 47 wheels of the bicycles and tricycles. How many of tricycle does this Cycle shop have? Write a program named Cycle.java and compute the total number tricycles at the shop.

1.15.13. FindX.java

Write a program named FindX.java to compute the number X based on the following formula:

$$5 + 19 + X + 47 = 194$$

1.15.14. MaleStudent.java

Assume that there are 389 students in a small middle school. There are 175 female students. Write a program named MaleStudent.java to compute how many students are male in this middle school.

1.15.15. Circle.java

Write a program named Circle.java that displays the area and perimeter of a Circle that has a radius of 9.5 using the following formula:

$$area = radius \times radius \times Math.PI$$
$$perimeter = 2 \times radius \times Math.PI$$

1.16. Do You Have Any Questions about Chapter 1?

Comments

2. Datatype, Variables, and Expressions

2.1. Learning Outcomes

- Understand the concept of variables.
- Understand the concept of data types in Java.
- Understand the syntax and semantics of data declaration statements and assignment statements.
- Compose arithmetic and String expressions.
- Use Scanner to get information from user.
- Print values and texts.

2.2. Key Terms

Review the important terms on Think Java Chapter 2 and Chapter 3.

2.3. Resources

Text

- Think Java by Allen Downey and Chris Mayfield. You can use the online interactive version or download the PDF version.
- Oracle Java Tutorial

Video/Instruction

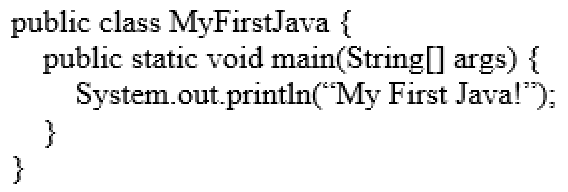
Converting a decimal number to floating-point format

2.4. Overview

Programming is to write code that can be executed by the computer to solve a particular problem. A program consists of instructions that a computer can execute. We call instructions written in a high-level programming language ***statements***. In this chapter, we will learn about statements that get input from user, perform computations, store data and computation results in memory and display messages.

The following Java program was first introduced in Chapter 1. In fact, it contains only one statement: ***System.out.println("My First Java!");***

MyFirstJava.java



```
public class MyFirstJava {
   public static void main(String[] args) {
      System.out.println("My First Java!");
   }
}
```

We can see that the program MUST contain other lines to conform to Java structure rules (i.e. syntax). The following is the class block or "wrap", containing the class header and class block (i.e. the pair of curly brackets).

```
public class MyFirstJava {

}
```

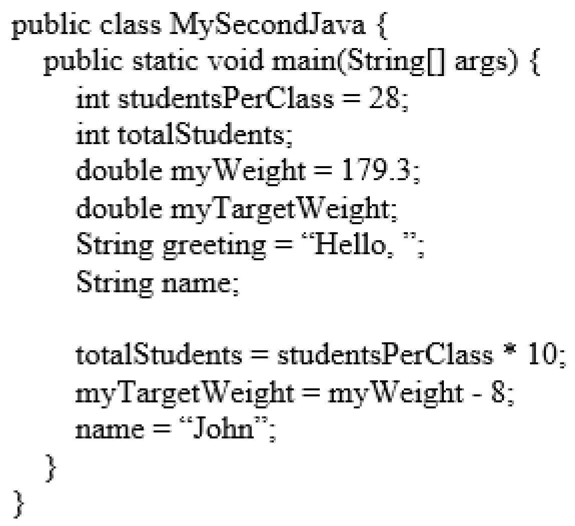
The following is the method block or "wrap", containing the method header and method block (i.e. the pair of curly brackets).

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

}
```

We can see that the method block is indented inside the class block. We can also see that the statement is further indented inside the method block. The indentation is for highlighting the structure of a program.

The following is another Java program that contains more statements. We will use it to illustrate some important Java concepts. We can see that the Java file name and the class name have to be the same. Java is case sensitive, which means that upper and lower cases have to match exactly.

MySecondJava.java



```
public class MySecondJava {
   public static void main(String[] args) {
      int studentsPerClass = 28;
      int totalStudents;
      double myWeight = 179.3;
      double myTargetWeight;
      String greeting = "Hello, ";
      String name;

      totalStudents = studentsPerClass * 10;
      myTargetWeight = myWeight - 8;
      name = "John";
   }
}
```

2.5. Variable

As introduced in Chapter 1, data are stored in memory. Instead of directly using memory addresses, Java uses variables. When the Java compiler translates a program, a variable is given a specific memory location (address).

In the above program ***MySecondJava.java***, ***studentsPerClass***, ***myWeight***, and ***greeting*** are variable names, each representing a memory location for storing data.

We want to choose meaningful names for variables, indicating the purpose of each variable. If the name you choose consists of only one word, spell that word in all lowercase letters, such as ***greeting***. If it consists of more than one word, capitalize the first letter of each subsequent word, such as ***studentsPerClass*** and ***myWeight***.

For more information, see this site: Java variable name rules and conventions

2.6. Data Type

A program can process different types of data. We will talk about three basic data types in Java: integer, real value, and text. Integers are whole numbers (without a decimal point). Real values are numbers that include decimal places. Texts are sequences of characters, including punctuation, symbols, and whitespace. Every value in Java has a corresponding data type. The table below shows examples of each of the three types.

Table 1. Basic Data Types

Data Type	Example Values	Java Data Types

integer	2, -2, 0, 834529	byte, short, int, long
real value	2.0, -2.235, 0.0, 8329.123782	float, double
text	"Hello World!", "Coconut", "0", "4 + 6"	String

Note that text data are always surrounded by quotes. Some text may look like numbers, but as long as they are surrounded by quotes, they are treated like text.

Java uses its special key words to represent these data types. There are four integer types.

- **byte**: 8-bit integer
- **short**: 16-bit integer
- **int**: 32-bite integer (most often used)
- **long**: 64-bite integer

Note: An integer value as in Table 1 is by default as the ***int*** type. To represent a ***long*** value, we need to append the value with an L or l (i.e. either upper or lowercase). For example, 2L and 23458907234556L are two long values.

The format to represent real values are called floating point. There are two floating-point data types:

- **float**: 32-bit floating point
- **double**: 64-bit floating point (most often used)

An example for converting a decimal number to float

If you are curious, you can watch this video for converting a decimal number to floating-point format. The example used in the video is 13.1875. You will see how it is converted to float.

Note that a real value as in Table 1 is by default the ***double*** type. To represent a float value, we need to append the value with an F or f (i.e. either upper or lowercase). For example, 2.0F and -2.235f are two values of the ***float*** type.

For text, the data type name is **String**. Note that, the type name is capitalized.

2.7. Declaration Statement: Declare Data Type for a Variable

In Java, each variable must be associated with a certain data type. We use declaration statement to specify the data type of a variable. For example, the following statement declares the variable ***studentsPerClass*** as an ***int*** variable.

```
int studentsPerClass;
```

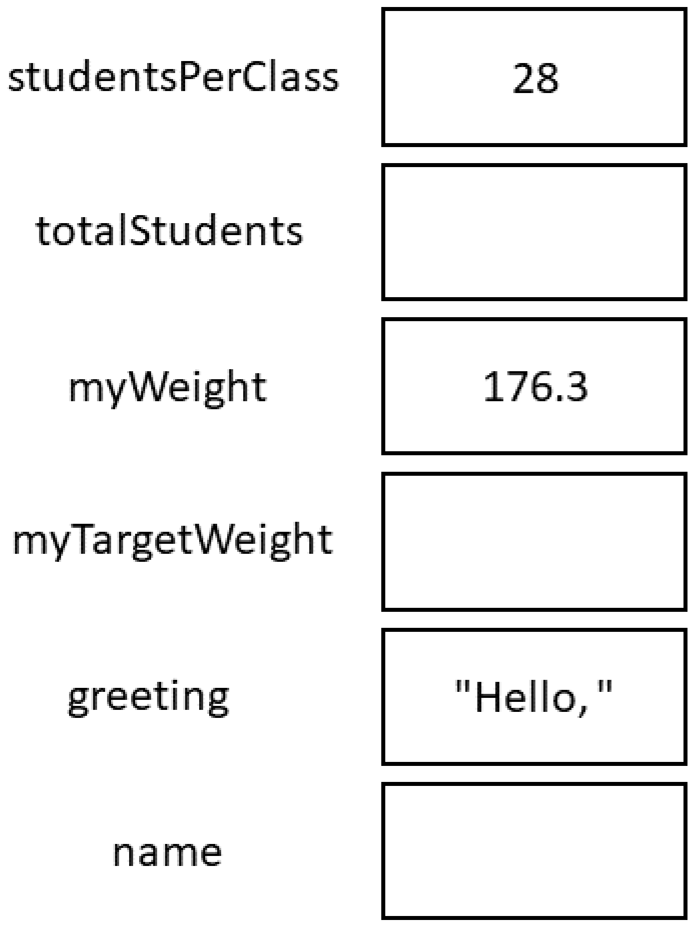
The following is also a declaration statement. Besides declaring the variable ***studentsPerClass*** as an ***int*** variable, it also initializes the variable with an int value 28.

```
int studentsPerClass = 28;
```

You can also declare multiple variables of the same type in one declaration statement.

```
int studentsPerClass = 28, totalStudents;
```

After all the declaration statements in ***MySecondJava.java*** are executed, the memory looks like below:



2.8. Assignment Statement: Change the Value of a Variable

The following statements are assignment statements. Note that they are similar to a declaration statement with an initial value, but there is no data type at the left. These variables have already been declared earlier, so no data types are needed on the left.

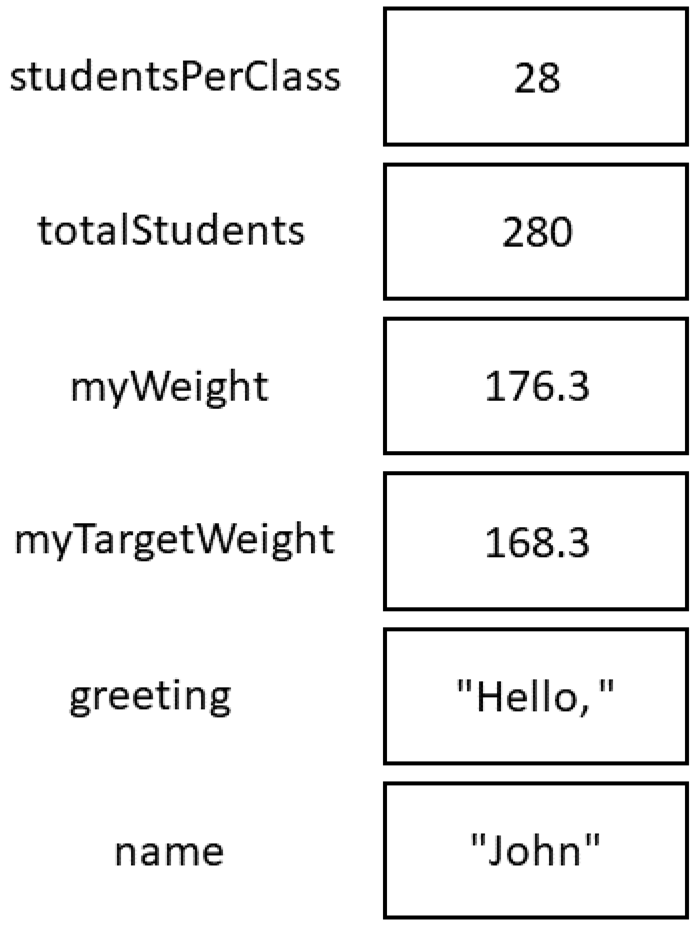
```
totalStudents = studentsPerClass * 10;
myTargetWeight = myWeight - 8;
name = "John";
```

An assignment statement puts a new value into a variable. The right side of an assignment statement is first evaluated, and the result will be put into the variable on the left.

The following describes what happens after each of the statements above is executed.

- Since ***studentsPerClass*** contains a value 28, ***studentsPerClass * 10*** will be 280. After the first assignment, the variable ***totalStudents*** is 280.
- Since ***myWeight*** contains a value 176.3, ***myWeight - 8*** will be 168.3. After the second assignment, the variable ***myTargetWeight*** is 168.3.
- The last assignment gives the variable ***name*** a value "John".

After these assignment statements are executed, the memory looks like below.



2.8.1. JavaTutor Example

JavaTutor Example (click "Vizualize Execution")

Note that if a variable is declared for a certain data type, trying to put a value of another incompatible type will cause a violation of syntax. For example, the compiler will report a syntax error for the following statement, since "168.3" is a String (text) and is not compatible with a double data type.

myTargetWeight = "168.3"; //This will cause a syntax error!

2.9. Expressions

An arithmetic expression (an expression for numeric values) is often used in an assignment statement. For example, ***studentsPerClass * 10*** and ***myWeight - 8*** are both arithmetic expressions. Please refer to Section 2.5 of Think Java for more information on this topic.

Java also has an operator, **%**, to get remainder of a division. Please refer to Section 3.8 of Think Java for more information on the remainder operator.

Further, Java also has String operations. Please refer to Section 2.8 Think Java for more information on String expressions. This section also talk about how String values and numeric values are "added" together.

For a more detailed introduction to floating-point numbers and the associated rounding errors, please read Section 2.6 and 2.7 of Think Java.

Finally, Java can convert a value of one data type to a value of another data type. For example, ***(int) 123.56*** converts a double value 123.56 to an int value 123. ***(int)*** is called a type cast operator. Please refer to Section 3.7 of Think Java for more information on this topic.

2.10. Output Statement

The current version of ***MySecondJava.java*** does not interact with the user who runs the program at all. Variable declarations and assignments all happen in memory. We will extend the program by adding output statements at the end. They communicate with the user by printing information on the screen.

MySecondJava.java

```
public class MySecondJava {
   public static void main(String [] args) {
      int studentsPerClass = 28;
      int totalStudents;
      double myWeight = 176.3;
      double myTargetWeight;
      String greeting = "Hello, ";
      String name;


      totalStudents = studentsPerClass * 10;
      myTargetWeight = myWeight - 8;
      name = "John";

      System.out.println(totalStudents);
      System.out.println(myTargetWeight);
      System.out.println(greeting + name + "!");
   }
}
```

The first output statement will print the value of variable ***totalStudents***, which is 280, on a line. The second output statement will print the value of the variable ***myTargetWeight***, which is 168.3, on the second line. The third output statement will print the result of the String expression ***greeting*** + ***name*** + ***"!"***, which is "Hello, John!" on the third line. Note that the quotation marks are not printed. The following is what the program will print:

```
280
168.3
Hello, John!
```

2.10.1. Formatting outputs using printf

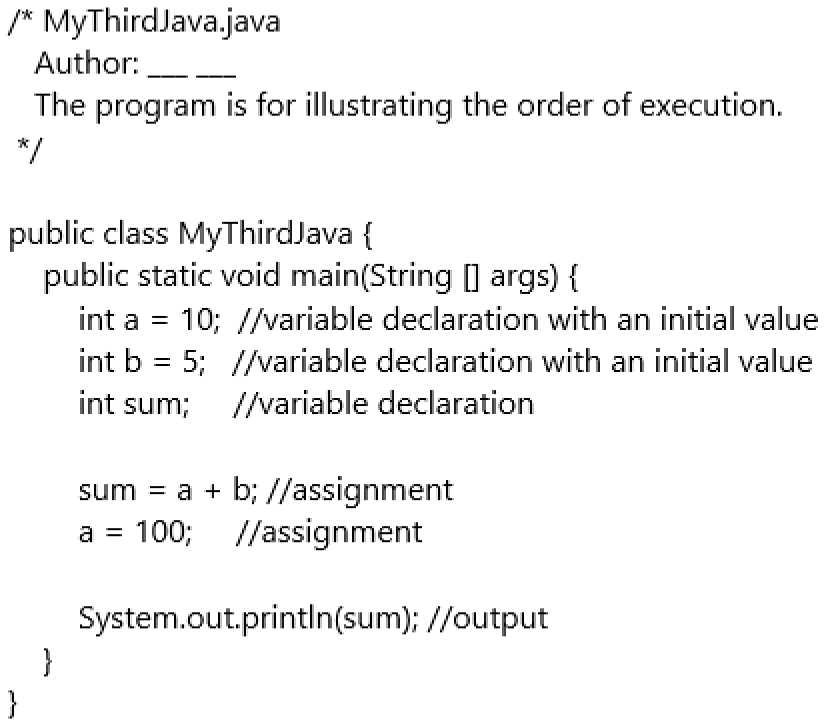
In addition to print and println, you can use ***System.out.printf*** to format the output. It's a very useful method and will be used in many of the exercises at the end of the chapter. Please read Section 3.5 of Think Java on how to have more control on output format.

2.11. Comments and Order of Execution

The following program contains comments. Comments are not instructions for the computer to follow, but instead notes for programmers to read. There are two types of comments in Java.

Line comments start with //. Anything following //, on the same line, will not be executed. Often, at the very beginning of a program, comments are used to indicate the author and other information. Even though line comments can be used, but for comments that span several line, we usually use block comments, which start with /* and end with **/. Anything in between /** and */ will not be executed. Comments are also used to explain tricky sections of code or disable code from executing.

MyThirdJava.java



```
/* MyThirdJava.java
   Author: ___ ___
   The program is for illustrating the order of execution.
 */

public class MyThirdJava {
   public static void main(String [] args) {
      int a = 10;  //variable declaration with an initial value
      int b = 5;   //variable declaration with an initial value
      int sum;     //variable declaration

      sum = a + b; //assignment
      a = 100;     //assignment

      System.out.println(sum); //output
   }
}
```

Now take a look at the program, can we give a guess as to what the program will print? Is it 15 or 105?

Some of you will guess 105. The reasoning might be the following: Even though a starts with 10, but it changes to 100. Since variable sum contains the result of a + b, so sum should be 105 at the end. However, this is INCORRECT.

The statements are executed in the order that they show up in the program. When line ***sum = a + b;*** is executed, sum gets the value 15. When line ***a = 100;*** is executed, a gets a new value 100. Note that sum is not changed and retains the value 15. The output statement thus prints 15.

2.12. Declare Constants and Get User Input with Scanner

Read the following program.

Miles100ToKMs.java

```
/* Miles100ToKMs.java
   Author: ___ ___
   The program is for illustrating the order of execution.
 */

public class Miles100ToKMs {
   public static void main(String [] args) {
      final double KMS_PER_MILE = 1.609; //declare a constant variable
      double miles = 100.0;


      double kilometers = miles * KMS_PER_MILE;


      System.out.println(miles + " miles = " + kilometers + "  kilometers");
   }
}
```

The first statement ***final double KMS_PER_MILE = 1.609;*** declares a constant variable ***KMS_PER_MILE***. Note that the keyword ***final*** is prepend to the declaration. For a constant variable, an initial value must be assigned to the variable at the declaration. Also note that, by convention, the name for a constant variable is all capital letters with underscores separating words. However, if a variable's name is all captial letters, it is not automatically a constant. The keyword ***final*** is needed to make a variable constant. Please refer to Section 3.4 of Think Java for more information on constants.

We can see that the program converts 100.0 miles into kilometers. First, variable ***miles*** gets a value 100.0. Then the variable ***kilometers*** gets a value 160.9. Last the output statement prints the value of the expression ***miles + " miles = " + kilometers + " kilometers"***.

Let's talk about ***miles + " miles = " + kilometers + " kilometers"***. It is a mixed-type expression and evaluated to ***"100 miles = 160.9 kilometers".*** Please refer to the note below and Section 2.8 Think Java for more information.

String and Numeric Values Mixed Operations

Variables ***miles*** and ***kilometers*** are double values, while ***" miles = "*** and ***" kilometers"*** are String values. Since the addition operator + is left-associative, the expression is evaluated from left to right. The following shows how the expression is evaluated:

1. First, ***miles*** + ***" miles = "*** is evaluated. When a numeric value is added to a String value, the Java compiler will insert code to automatically convert the numeric value to the corresponding String value. In this case, ***miles***'s value 100.0 is converted to ***"100.0"***. Then ***"100.0"*** + ***" miles = "*** will be performed and results in a longer String ***"100 miles = "***.
2. Next, ***"100 miles = "*** + ***kilometers*** will be evaluated. It is a mixed-type operation and ***kilometers***'s value 160.9 is converted to "160.9". ***"100 miles = "*** + ***"160.9"*** will be carried out and results in ***"100 miles = 160.9"***.

Last, ***"100 miles = 160.9"*** + ***" kilometers"*** will be evaluated. Both values besides the + operator are String values and they are concatenated, resulting in ***"100 miles = 160.9 kilometers"***.

Now we understand the program: It converts 100 miles to the corresponding kilograms. However, this program has a big drawback. What is it?

This program can ONLY convert 100.0 miles. If there is a need to convert 200.0 miles, the programmer will need to modify 100.0 to 200.0. This basically changes the program itself and the program will need to be recompiled. A user will need to get this new program for converting 200.0 miles.

The following is a program that can get user input for distance in miles and then convert it into the corresponding kilometers. There is no need to modify and recompile the program for converting different values. The program can be run as many times as needed to convert whatever the values to be converted.

MilesToKMs.java

```
/* MilesToKMs.java
   Author: ___ ___
   The program is for illustrating the order of execution.
 */

public class MilesToKMs {
   public static void main(String [] args) {
      final double KMS_PER_MILE = 1.609; //declare a constant variable

      java.util.Scanner  input = new java.util.Scanner(System.in);
      System.out.print("Enter the distance in miles: ");
      double miles = input.nextDouble();


      double kilometers = miles * KMS_PER_MILE;


      System.out.println(miles + " miles = " + kilometers + " kilometers");
   }
}
```

The following are three new lines in the above program:

```
java.util.Scanner  input = new java.util.Scanner(System.in);  //new line 1
System.out.print("Enter the distance in miles: ");           //new line 2
double miles = input.nextInt();                             //new line 3
```

The second line is an output statement that prints a prompt for a user. It tells the user that the program expects a distance in miles to be entered. We will focus on the other two lines. We will talk about the first and third line below.

2.12.1. Create a Scanner Object

Line 1 ***java.util.Scanner input = new java.util.Scanner(System.in);*** looks complicated, but a closer look tells us that it's a declaration statement with an initial value assigned to the declared variable. The data type is ***java.util.Scanner***, the variable is ***input***, and the initial value is ***new java.util.Scanner(System.in);***.

The data type ***java.util.Scanner*** looks very long, but it really is ***Scanner.*** We will rewrite the program above as follows:

MilesToKMs.java

```
/* MilesToKMs.java
   Author: ___ ___
   The program is for illustrating the order of execution.
 */
import java.util.Scanner;

public class MilesToKMs {
   public static void main(String [] args) {
      final double KMS_PER_MILE = 1.609; //declare a constant variable

      Scanner input = new Scanner(System.in);
      System.out.print("Enter the distance in miles: ");
      double miles = input.nextDouble();


      double kilometers = miles * KMS_PER_MILE;


      System.out.println(miles + " miles = " + kilometers + " kilometers");
   }
}
```

We used the import statement ***import java.util.Scanner;*** to tell the Java compiler that data type Scanner is defined in the package ***java.util*** of the Java installation. A package contains multiple related classes. ***java.util.Scanner;*** is the full pathname of the data type Scanner. If we use the import statement, there is no need to use the full name in the remainder of the program. As a result,

```
java.util.Scanner input = new java.util.Scanner(System.in);
```

is simplified to

```
Scanner input = new Scanner(System.in);
```

Word ***new*** is a keyword for creating a new object. The expression ***new Scanner(System.in)*** creates a Scanner object that will extract information from ***System.in*** (see the note below). This object is assigned to the variable ***input***. Note that when an object is assigned to a variable, the reference (or location) of the object is stored in the variable.

Scanner, System, System.out, and System.in

Scanner *is a data type defined in class Scanner (in file Scanner.java). Similarly, the data type* **String** *is also a class defined* **String.java**__ file. The reason that we don't need an import statement for String is because String belongs to the **java.lang** package, which is imported automatically.

System is another class in the ***java.lang*** package. It is a class that provides methods related to the "system" or environment where programs run.

System.out is an object contained in the ***System*** class. It represents the standard output device, normally the monitor.

System.in is an object representing the standard input device, normally the keyboard.

2.12.2. Invoke Methods on the Scanner Object to Retrieve Information

Remember that we can invoke the ***print*** and ***println*** methods on ***System.out*** by ***System.out.print(…)*** or ***System.out.println(…)*** to print information.

Similarly, we can also invoke different methods on a Scanner object to retrieve different types of information. For example, ***input.nextInt()*** will invoke the nextInt method on the Scanner object ***input*** and retrieves an integer. See Table 2 for more Scanner methods.

Table 2: Scanner Methods

Method	Description
nextInt()	retrieve an int value
nextDouble()	retrieve a double value
next()	retrieve a word (a word is a sequence of characters with no space or tab or newline)
nextLine()	retrieve a line of characters

When the statement ***double miles = input.nextDouble();*** is executed, the program will wait and allow the user to enter data. Whatever the user enters will be retrieved and returned by the ***nextDouble*** method and then assigned to the variable ***miles***.

See the step-by-step walkthrough of this program at Java Visualizer.

Note that when using ***nextLine*** and the other next methods (e.g. ***next***, ***nextDouble***, ***nextInt***) together, there is an unexpected behavior. Please read Section 3.9 of Think Java to understand this phenomenon and how to deal with this issue in your program.

2.13. Augmented Assignment Operators

Java designers want to make life easier for Java developers and they designed some short-cut assignment operators to save typing time.

Let's look at the following three statements. You might say, "Hi, x cannot be equal to x + 5!"

$$x = x + 5$$

We need to remember, however, that this is not a math equation, but an assignment statement. The right side of an assignment statement, which is always an expression, is first evaluated. The value of the expression will be assigned to the variable on the left.

Assume ***x***'s original value is 10, ***x + 5*** is evaluated to 15 and then 15 will be put into ***x***. After the assignment statement is executed, ***x***'s value becomes 15.

For assignment statements that have similar format, Java has a set of augmented assignment operators. For example, the statement on the left of each line below is equivalent to the statement on the right. The operators used in the statements on the right side are the ***augmented assignment operators*** ***+=, -=, *=, /=, %=***. Note that there is no space inside the operators.

```
x = x + 5; <==> x += 5;
x = x - 5; <==> x -= 5;
x = x * 5; <==> x *= 5;
x = x / 5; <==> x /= 5;
x = x % 5; <==> x %= 5;
```

Note that the format of the original assignment statement has to be the following, where both <var>'s refer to the same variable.

```
<var> = <var> <operator> <expression>
```

Some Tricky Points

It's tricky when <expression> is more complicated than a single item. For example,

```
x = x - 5 + 4;
```

is not equivalent to

```
x -= 5 + 4; //WRONG
```

Instead, we need to convert the original assignment statement to the following before converting it using the augment assignment operator +=.

```
x = x - (5 - 4);
```

and then it can be safely converted to the following equivalent statement:

```
x -= 5 - 4; //CORRECT
```

If a variable is increased by 1 or reduced by 1, there are ways to further save typing. We can use ***increment operator*** ++ and ***decrement operator --***. On each line, all statements are equivalent.

```
x = x + 1; <==> x += 1; <==> x++; <==> ++x;
x = x - 1; <==> x -= 1; <==> x--; <==> --x;
```

Post/Pre Incrementation/Decrementation

There are some differences between the following pairs, if you used increment or decrement operators inside another statement.

```
x++ (post-incrementation) v.s. ++x (pre-incrementation)
x-- (post-decrementation) v.s. --x (pre-decrementation)
```

For x++, the value of x is used before x is incremented. For example, for the following two statements, x's value 10 first assigned to y and then x is incremented to 11, so after both statements are executed, x is 11 and y is 10. .

```
x = 10;
y = x++;
```

For ++x, x is incremented first before the value of x is used. For example, after for the following two statements, x is first incremented to 11 and then its value 11 is assigned to y, so after both statements are executed, x is 11 and y is 11.

```
x = 10;
y = ++x;
```

We will not dive too deep into this topic here. We believe that it makes code difficult to read and, therefore, we discourage this practice.

2.14. Exercises

2.14.1 Exercise 1

Write a program to convert 100 Fahrenheit to the corresponding temperature in Celsius. User proper variable names for the temperature in Fahrenheit and thee temperature in Celsius. Include a proper arithmetic expression to do the conversion.

2.14.2 Exercise 2

Write a program that does the following:

a. creates variables named day, date, month, and year. The variable day will contain the day of the week (like Friday), and date will contain the day of the month (like the 13th).
b. Assign values to those variables that represent 2019 Labor Day's date (9/2/2019, Monday).
c. Display the value of each variable on a line by itself. This is an intermediate step that is useful for checking that everything is working so far. Compile and run your program before moving on.
d. Finally, modify the program so that it displays the date in standard American format, for example: Thursday, July 16, 2015. 5. Modify the program so it also displays the date in European format.

   American format:
   Thursday, July 16, 2015
   European format:
   Thursday 16 July 2015

The final output should be in the following format. Note that the actual date should be for 2019 Labor Day.

2.14.3 Exercise 3

The point of this exercise is to (1) use some of the arithmetic operators, and (2) start thinking about compound entities (like time of day) that are represented with multiple values.

a. Create a new program called Time.java. From now on, we won't remind you to start with a small, working program, but you should.
b. Create variables named hour, minute, and second. Assign values to these variables that represent the time 30 seconds after 5:15pm. Use a 24-hour clock. For example, for 2pm, the value of hour is 14. Make the program calculate and display the number of seconds since midnight.
c. Calculate and display the number of seconds remaining in the day.
d. Calculate and display the percentage of the day that has passed. You might run into problems when computing percentages with integers, so consider using floating-point.

Hint: You might want to use additional variables to hold values during the computation. Variables that are used in a computation but never displayed are sometimes called "intermediate" or "temporary" variables.

2.14.4 Exercise 4

Write a program that converts a temperature from Celsius to Fahrenheit. It should

a. prompt the user for input,
b. read a double value from the keyboard,
c. calculate the result,
d. format the output to one decimal place using printf.

Here is the formula. Be careful not to use integer division! F = 9/5 C + 32

2.14.5 Exercise 5

Write a program that calculates the base area and volume of a cylinder, given the radius of the base circle and the length of the cylinder. Please use 3.1416 as the PI value.

2.14.6 Exercise 6

Write a program that convert a given total number of seconds into hours, minutes and seconds. It should (1) prompt the user for input, (2) read an integer from the keyboard, (3) calculate the result, and (4) display the output.

The following are two examples:

- if the total number of seconds is 125, it is 0 hours, 2 minutes and 5 seconds.
- If the total seconds is 7450, it is 2 hours, 4 minutes and 10 seconds.

2.14.7 Exercise 7

Given the length, width and height of the room, number of windows and number doors, the program will calculate the gallons and quarts of paint are needed to paint the room. For gallons, print an integer value. For the quarts, print a real number, no need to round down to integer.

Assume that (1) ceiling is painted, (2) 1 gallon of paint covers about 350 square feet, (3) each window is 15 square feet, and (4) each door is 21 square feet.

2.14.8 Exercise 8

Write a program that prompts the user to enter a number between 0 and 1000 (not including 1000) and print the digits in reverse order. The following are some examples:

- For a given value 256, the program should print 652.
- For 23, since the digit at hundreds is 0, the program should print 320.
- For 10, since the digit at ones and hundreds are 0, the program should print 010.
- For 0, the digits at ones, tens and hundreds are all 0, so the program should print 000.

2.14.9 Exercise 9

Write a program that prompts the user to enter a monthly saving amount and displays the account value after each of the first six months. When displaying the account values, please use printf to print only two decimal places.

We will assume that the yearly interest is 5%. Please declare a constant variable for the monthly interest rate as follows in your program:

final double MONTHLY_RATE = 0.05/12;

The following are how to calculate the account value at the end of each month:

*month1Value: monthlySaving * (1+MONTHLY_RATE)*

*month2Value: (monthly1Value + monthlySaving) * (1+MONTHLY_RATE)*

*month3Value: (monthly2Value + monthlySaving) * (1+MONTHLY_RATE)*

and so on.

2.14.10 Exercise 10

Write a program to calculate how many mile, feet, and inches a person walks a day, given the stride length in inches (the stride length is measured from heel to heel and determines how far you walk with each step), the number of steps per minute, and the minutes that person walks a day. Note that 1 mile is 5280 feet and 1 foot is 12 inches. In your program, there should be the following constants:

final int FEET_PER_MILE = 5280;

final int INCHES_PER_FEET = 12;

*final int INCHES_PER_MILE = FEET_PER_MILE * INCHES_PER_FEET;*

Hints: In the computation step, your program should first calculate the total inches that person walks a day. Then convert it to miles, remaining feet, and remaining inches. All variables should be of int type.

2.14.11 Exercise 11

Write a program that prompts user to enter the weight in pounds (real value) and height in feet and inches (both integers) and then calculates the BMI of the person. Please limit the print out of the BMI value to two decimal digits using printf.

Note that

> *1 kilogram is 2.2 pounds.*
> *1 inch is 0.0254 meters.*

Your program first need to convert pounds to kilograms (i.e. pounds/2.2) and inches to meters (i.e. inches * 0.0254) and then use the following formula to calculate BMI:

> *kilogram / (meter)^2*

2.14.12 Exercise 12

Write a program to calculate the cost of a trip. The program prompts user to enter the distance in miles of the trip, the car efficiency in miles per gallon, and the cost of gas in dollars per gallon. Please limit the print out of the cost to two decimal digits using printf.

2.15. Do You Have Any Questions about Chapter 2?

Comments

3. Conditions

3.1. Learning Outcomes

- Learn about boolean Datatype.
- Understand relational operators and logical operators.
- Understand how to evaluate boolean expressions (conditions) and compound boolean expressions.
- Use conditional statement (if-statement), multibranch (ladder) if statement and nested if statement to solve common problems.
- Write switch statement and use it to write programs.

3.2. Key Terms

Review important terms.

3.3. Resources

- Java Documentation, the java tutorial, The if-then and if-then-else Statements
- Think Java, Chapters 5: How to Think Like a Computer Scientist by Allen Downey and Chris Mayfield Skip Recursion Sections(5.8 – 5.10)
- Safari, Flow Control Structure Video
- Lynda.com, “Workig with boolean Values and Expressions”
- Lynda.com, “Programming Conditional Logic”
- Lynda.com, “Using the switch Statement”
- Lynda.com, “Java 8 Essential Training”

3.4. Overview

In real life, we often encounter situations where we need to make decisions based on certain conditions. For example, based on the student numerical grade we want to determine whether the student is passing or failing the class.

So far, the programs we have been writing do not support different outcomes such as determining whether a student is passing or failing. They do not support computations that depend on the conditions, and they cannot validate user input. Our programs need to produce different results based on conditions encountered in the program.

In this chapter we introduce the boolean data type, a primitive type that can hold true or false. We discuss relational and logical operators and build an understanding of boolean expressions (conditions), and introduce conditional statements in Java (if, switch and the ternary expression).

By the end of this chapter, you should be able to use boolean expression and conditional statement effectively to solve common programming problems. It is important to note that the terms boolean expression and condition are used interchangeably throughout this chapter.

3.5. Relational Operators, Simple Boolean Expressions and the boolean Data type:

We will start this section by introducing the boolean data type. A variable of boolean data type can have only one of two values, either true or false. The example below shows how to declare and initialize a boolean data type:

```
boolean x;
x = true;
```

Or alternatively, the two steps can be combined in one statement as below:

```
boolean x = true;
```

Relational operators are used to check conditions. They can be used to check whether two values are equal, not equal, greater than or less than, the table below list the java different relational operators along with their mathematical equivalent operators, the result of an operational operator is either true or false (boolean):

Table 3. Relational Operators

Java	Mathematics
==	=
!=	≠
>	>
<	>
>=	≥
<=	≤

Note: == is used to check for equality

Boolean expressions use relational operators to check conditions and to compare variables against certain values. For example, in order for a student to pass a class his grade must be greater than or equal to 70; that is, we need to compare the student grade with 70. The boolean expression (condition) that represents this comparison is **grade >= 70**. A boolean expression will evaluate to either true or false. The figure below shows some examples of boolean expressions:

```
int i = 5;
int j = 10;
boolean b;

i == j; //Evaluates to false, because i is not equal to j
i < j;  //Evaluates to true, because i is less than j
i <= j; //Evaluates to true, because i is less than or equal to j

b = i > j; //the value assigned to b is false
b = i != j; //the value assigned to b is true
b = j >= i; //the value assigned to b is true
```

3.6. Logical Operators and Compound Boolean Expressions

Sometimes we need to check for two or more conditions in order to decide or pick an execution path. Logical operators allow us to combine two more conditions. A compound boolean expression consists of two or more boolean expressions joined with logical operators. For example, assume we have a class x that requires two prerequisites class A and class B, in order to decide whether a student can register for class x or not, we need to verify that the student has passed both class A and class B. In this section we will cover the logical and operator (&&), the logical *or* operator (||), the logical *negation* (*not*) operator (!) and *exclusive or* operator "XOR" (^). the four logical operators with their Java symbols are shown in the table below, ***and***, ***or*** and ***exclusive or*** are binary operators, that is they take two operands while ***not*** is a unary operator, that is, it has only one operand:

Table 4. Logical Operators

Logical Operator	Java Symbol
and	&&
or	||
not	!
XOR	^

The table below shows the truth table for &&, ||, ^ and !, where b1 and b2 stands for boolean expression 1 (condition1) and boolean expression 2 (condition2), respectively:

Table 5: Truth Table

b1	b2	b1 && b2	b1 || b2	b1 ^ b2	!b1
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

Referring to the table above we notice the following:

- && result is true only if both boolean expression b1 and boolean expression b2 are true, otherwise, && result is false.
- || result is false only if both boolean expression b1 and boolean expression b2 are false, otherwise, || result is true.
- ^ result is false if both conditions are true or both conditions are false, otherwise ^ result is true. That is ^ result is true if both boolean expressions have different values.

Below are examples of compound boolean expressions:

```
int i = 5;
int j = 10;
int k = 15;

//&&
i < j && j < k; //true: both i < j and j < k are true
i > j && j < k; //false: i > j is false
i < j && j > k; //false: j > k is false
i > j && j > k; //false: both i > j and j > k are false

//||
i < j || j < k; //true: both i < j and j < k are true
i > j || j < k; //true: j < k is true
i < j || j > k; //true: i < j is true
i > j || j > k; //false: both i > j and j > k are false

//^
i < j ^ j < k; //false: both i < j and j < k are true
i > j ^ j < k; //true: i >j is false and j < k is true
i < j ^ j > k; //true: i < j is true and j > k is false
i > j ^ j > k; //false: both i > j and j > k are false

//!
!(i < j); //false: i < j is true
!(i > j); //true: i > j is false
```

Java use short circuit evaluation for both && and || operators. In the case of &&, if the first boolean expression (condition) evaluates to false, the result of && is false regardless of whether the second boolean expression evaluates to true or false. In the case of ||, if the first boolean expression (condition) evaluates to true, the result of || is true regardless of whether the second boolean expression evaluates to true or false. To better understand short circuit analysis, assume we have two boolean expressions b1 and b2. In the case of && (b1 && b2), assume b1 is false, java will not evaluate b2 since, since b1 && b2 is false regardless the value of b2. In the case of || (b1 || b2), assume b1 is true, java will not evaluate b2 since, since b1 || b2 is true regardless the value of b2.

3.7. Operator Precedence

The table below show the operator Precedence from highest to lowest:

Table 6: Operator Precedence

Precedence
var++, var--
Unary +, Unary -, ++var, --var
!
*, / , %
+ (addition), - (subtraction)
<, <=, >, >=
^
&&
||
=, +=, -=, *=, /=, %=

3.8. if Statement

In the introduction section, we presented the problem of determining whether a student is passing a class or not based on the student numerical grade; assuming a passing grade of 70, a student is passing the class if his/her grade is greater than or equal to 70. The java if statement is needed to solve this problem. In this section, we will introduce if, if-else, nested if, and multibranch if-else statements.

3.8.1. if Statement

The general syntax for an if statement is java is:

```
if (boolean expression/condition) {
        Statement1;
        Statement2;
        .
        .
        .
}
```

The statements inside the if block are executed only if the boolean expression evaluates to true, otherwise the entire block will be skipped. Notice that, the curly braces ({}) are needed only if we have more than one statement that need to be executed if the boolean expression evaluates to true.

Example 1: Write a program that prompts the user to enter his grade, the program evaluates the grade and prints *Passing* if the student is passing the class and prints *Not Passing* if the student is not passing the class.

Step-by-Step Execution for the program below

```
//This program prompts the user to enter his grade
//Based on the entered grade the program will print
//passing or not passing
import java.util.Scanner;

public class CheckGrade{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter your grade: ");
    double grade = in.nextDouble();
    if (grade >= 70){
      System.out.println("Passing");
    }
    if (grade < 70){
      System.out.println("Not Passing");
    }
  }
}
```

Example 2: Write a program that prompts the user for an integer value and then prints if the entered value is odd or even.

Step-by-Step Execution for the Program Below

```
/*
This program prompts the user to enter an integer
and prints if it is odd or even
% is used to determine whether the integer is odd or even
*/

import java.util.Scanner;

public class OddEven{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter an integer value: ");
    int i = in.nextInt();
    if (i % 2 == 0){
      System.out.println(i + " is an even number.");
    }
    if (i % 2 != 0){
      System.out.println(i + " is an odd number.");
    }
  }
}
```

Example 3: Write a program that reads a student numerical grade and prints the letter grade of the student, grade >= 90, prints A, 90 > grade >= 80, prints B, 80 > grade >= 70, prints C, 70 > grade >= 60, prints D, grade < 60, prints F. This example illustrates the use of compound boolean expressions.

Step-by-Step Execution of the Program Below

```
/* This program prompts the user to enter his grade
Based on the entered grade the program will print
letter grade, A, B, C, D, or F */
import java.util.Scanner;

public class LetterGrade{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter your grade: ");
    double grade = in.nextDouble();
    if (grade >= 90)
      System.out.println("Your letter grade is A");
    if (grade < 90 && grade >= 80)
      System.out.println("Your letter grade is B");
    if (grade < 80 && grade >= 70)
      System.out.println("Your letter grade is C");
    if (grade < 70 && grade >= 60)
      System.out.println("Your letter grade is D");
    if (grade < 60)
      System.out.println("Your letter grade is F");
  }
}
```

3.8.2. if-else Statement

Considering example 1 above, we notice that there is no need to evaluate the boolean expression again, since a student can only be passing or not passing. Checking the first boolean expression is enough to decide whether the student is passing or not passing, Therefore, evaluating the second boolean expression is not needed. The same discussion is applicable to second example. To avoid redundant evaluations java introduces an if-else statement. The general syntax for an if-else statement is java is:

```
if (boolean expression/condition) {
    Statement/statements if boolean expression evaluates to true;
} else{
    Statement/statements if boolean expression evaluates to false;
}
```

Note: There is no boolean expression that follow else.

The statements inside the if block are executed only if the boolean expression evaluates to true, otherwise the statements in the else block will be executed.

Example 4: rewrite the program from example 1 using an if-else statement instead of two if statements, the program prompts the user to enter his grade and will print either, he is passing the class, or he is not passing the class.

Step-by-Step Execution of the Program Below

```
//This program prompts the user to enter his grade
//Based on the entered grade the program will print
//passing or not passing
import java.util.Scanner;

public class CheckGradeV2{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter your grade: ");
    double grade = in.nextDouble();
    if (grade >= 70)
      System.out.println("Passing");
    else
      System.out.println("Not Passing");
  }
}
```

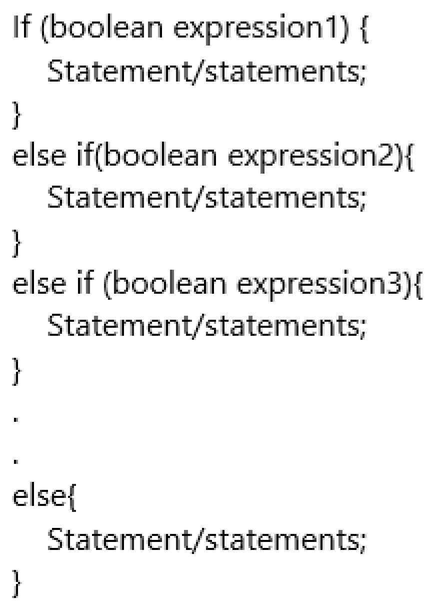
Example 5: rewrite the program from example 2 using an if-else statement instead of two if statements, the program that prompts the user for an integer value and then prints if the value is odd or even.

Step-by-Step Execution of the Program Below

```
//This program prompts the user to enter an integer
//and prints if it is odd or even
// % is used to determine whether the integer is odd or even

import java.util.Scanner;
public class OddEvenV2{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter an integer value: ");
    int i = in.nextInt();
    if (i % 2 == 0)
      System.out.println(i + " is an even number.");
    else
      System.out.println(i + " is an odd number.");
  }
}
```

3.8.3. Multi-branch (Cascaded) if Statement

In example 3, we wrote a program that read a student's numerical grade and printed the letter grade. Analyzing the program, we found, even though we know the assigned letter grade, we continued checking the subsequent conditions. For example, if the student grade was 90, evaluating the first boolean expression (grade >= 90) will result in letter grade of A and we should stop, however, we noticed that the program will continue evaluating other subsequent statements. These subsequent evaluations are unnecessary. To avoid such redundancy, a multibranch if statement (if – else if – else) can be used. Example 6 shows the solution for example 3 using a multibranch if statement. It is important to note that only one branch of the multibranch if statement is executed. The general syntax for a multi-branch (cascaded) if statement in java is:



```
If (boolean expression1) {
   Statement/statements;
}
else if(boolean expression2){
   Statement/statements;
}
else if (boolean expression3){
   Statement/statements;
}
.
.
else{
   Statement/statements;
}
```

Example 6: revisiting example 3, Write a program that reads a student numerical grade and prints the letter grade of the student, grade >= 90, prints A, 90 > grade >= 80, prints B, 80 > grade >= 70, prints C, 70 > grade >= 60, prints D, grade < 60, prints F.

Step-by-Step Execution of the Program Below

```
//This program prompts the user to enter his grade
//Based on the entered grade the program will print
//letter grade, A, B, C, D, or F
import java.util.Scanner;

public class LetterGradeV2{
  public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter your grade: ");
    double grade = in.nextDouble();
    if (grade >= 90)
      System.out.println("Your letter grade is A");
    else if (grade >= 80)
      System.out.println("Your letter grade is B");
    else if (grade >= 70)
      System.out.println("Your letter grade is C");
    else if (grade >= 60)
      System.out.println("Your letter grade is D");
    else
      System.out.println("Your letter grade is F");
  }
}
```

3.8.4. Nested if Statement

A nested if statement is an if statement that embedded inside another if or else statement. The general syntax for a nested if statement in java is:

```
if (boolean expression1) {
    //executes if boolean expression1 evaluates to true
    if (boolean expression2){
        statement/statements to be executed if boolean expression2 is true;
    }
}
```

Example 7: Write a program that reads an integer, and prints “You Won!!!” if the entered value is between 50 and 100 inclusive.

Step-by-Step Execution of the Program Below

```
/* This Program prompts the user to enter an integer value
 * and prints "You Won!!!" if the entered value is between
 * 50 and 100 inclusive.
 */
import java.util.Scanner;
  public class NestedIfExample{
    public static void main(String[] args){
    Scanner in = new Scanner(System.in);
    System.out.print("Please enter an integer: ");
    int i = in.nextInt();
    if (i <= 100)
      if (i >= 50)
        System.out.println("You Won!!!");
    }
  }
```

3.9. switch Statement

A switch statement is multi-branch statement, in which the execution path is based on a value of a variable or an expression. Based on the java documentation, the expression or the variable can be byte, short, char and int primitive data types. In addition, it works with String, Character, Byte, Short, Integer, etc. Unlike the if statement, a switch statement can have several execution paths. The Syntax for the switch is shown below, where x and y are values and are not variables. Note that having a default block is optional:

```
switch(expression) {
    case x:
        statement/statements
        break;
    case y:
        statement/statements
        break;
    .
    .
    default:  //optional
        statement/Statements
}
```

The switch statement works as follow:

- The switch expression is evaluated only once.
- The expression value is compared to the value of each case, if there is a match the code under that case is executed until a break statement is reached, or the end of switch block is reached. If there is no match the block under default case is executed if a default label exists. Notice the default case is the last case in a switch statement.

Example 7: Write a program that reads the year and month as integers and print the number of days in the entered month. You can assume 1 for January, 2 for February, and so on.

Solution: For January, March, May, July, August, October and December (months 1, 3, 5, 7, 8, 10, 12) all have 31 days. For April, June, September and November (months 4, 6, 9, 11) all have 30 days. For February (month 2) will depend on the year. If the year is leap, it will have 29 days otherwise it will have 28 days.

Step-by-Step Execution for the Program Below

```
/*
This program prompts the user for the month and the year
and prints the number of days in the entered months
Month and year should be entered as integer
*/
import java.util.Scanner;
public class NumberOfDaysInMonth{
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        //read year as integer
        System.out.print("Please enter year: ");
        int year = in.nextInt();
        //read month
        System.out.print("Please enter a month 1 - 12: ");
        int month = in.nextInt();
        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                System.out.println("Number of days of month " + month + " is " + 31);
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                System.out.println("Number of days of month " + month + " is " + 30);
                break;
            case 2:
                if (((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0))
                   System.out.println("Number of days of month " + month + " is " + 29);
                else
                  System.out.println("Number of days of month " + month + " is " + 28);
                break;
            default:
                System.out.println("Invalid month.");
                break;
        }
    }
}
```

3.10. Summary

In this chapter, we covered the boolean datatype which can store true or false values. We also learned about the relational operators (<, <=, >, >=, ==, !=) and studied boolean expressions the yield either true or false. We also studied the logic operator (&&, ||, ! and ^) and how they are used to combine boolean expressions (conditions) to produce compound boolean expressions. Then we studied the if statement and discussed the need for this statement to write programs that allow us to make decisions based on certain criteria. Finally, we covered the switch statement.

3.11. Exercises/Problem Solving:

Group 1: Exercise 1-7 of Section 5.11 of Think Java PDF or read the problem descriptions online at Section 5.11

Group 2:

3.11.1 Exercise 1

Write a program that generates a random integer between 0 and 10 and ask the user to guess the generated number. if the user enters the correct number, the program will print *Hooray you guessed the number*. Otherwise the program prints *You Lost!!*. Hint: you can use Math.random() to generate a random number as follows:

```
int rand = (int)(Math.random() * 11);
```

3.11.2 Exercise 2

Write a program that prompts the user to enter the radius of a circle and calculates the area for that circle. The program should check the entered number. if the entered number is negative, the program prints *Invalid Entry, the radius should be positive* and quits. Otherwise, the program should calculate the area of the circle and prints *The area of a circle with radius "the radius value entered by the user" is "The calculated value of the circle area"*. (Hint: Circle area = π * radius * radius)

3.11.3 Exercise 3

Write a program the reads three integers from the user and prints the largest number.

3.11.4 Exercise 4

Write a program that prompts the user to enter a number and check the following:

- if the number is divisible by 3 and 5, the program prints the number is a multiple of 3 and a multiple of 5

- if the number is divisible by 3 or 5, the program prints the number is a multiple of either 3 or 5.
- if the number is divisible by only one of the numbers, the program prints the number is divisible by either 3 or 5 but not both.

3.11.5 Exercise 5

Write a program that prompts the user to enter the day of the week as an integer between 1 and 7, for Sunday through Saturday, and prints weekday for entries 1 through 5 inclusive, and weekend for 6 and 7. for all other entries, the program prints invalid weekday.

3.11.6 Exercise 6

Write a program to calculate the cost of car insurance based on the driver age and number of accidents. The base insurance cost is $300. if the driver age is below 27, there is a surcharge of $100. the additional surcharge for accidents is shown below:

Surcharge Per accidents

Number of accidents	*Surcharge*
1	*$100*
2	*$150*
3	*$250*
4 or more	*$1000*

3.11.7 Exercise 7

Write a Program that prompts the user to enter four integers and prints them in alphabetical order.

3.11.8 Exercise 8

Write a program that prompts the user to enter the length of the three edges of a triangle. The program calculates the perimeter of the triangle if the input is valid, otherwise it prints *invalid input*. The input is valid if the sum of every pair of two edges is greater than the remaining edge.

3.11.9 Exercise 9

Write a program that computes and interprets the Body Mass Index (BMI). The program prompts the user to enter his/her weight in pounds and his/her height in inches. the program then calculates the bmi using the formula: BMI = Weight(kilograms)/(height(meters))^2. To convert weight in pounds(p) to kilograms(kg) use the formula: weight(kg) = weight(p) * 0.4536. To convert inches(in) into meters(m) use the formula: height(m) = height(in) * 0.0254. The BMI interpretation is as follows:

BMI Interpretation

BMI	***Interpretation***
BMI < 18.5	*Underweight*
18.5 ≤ BMI < 25.0	*Normal*
25.0 ≤ BMI < 30.0	*Overweight*
BMI ≥ 30.0	*Obese*

3.12. Do You Have Any Questions about Chapter 3?

Comments

4. Loops

4.1. Learning Outcomes

Students will be able to:

a. Write different types of loops, such as for, while and do-while
b. Select the correct type of loop based on a given problem
c. Use loops to solve common problems, such as, finding sum, finding max, etc.
d. Write nested loops
e. Use the keywords break and continue

4.2. Key Terms

Review the important terms.

4.3. Resources

4.3.1. Text

- Think Java : How to Think Like a Computer Scientist by Allen Downey and Chris Mayfield
- Think Java Chapter 7: Loops. **Note:** The topic of recursion will not be covered in this class. Students should ignore the reference to recursion in this linked chapter.
- Java for Absolute Beginners : Java for Absolute Beginners

4.3.2. Videos

- Java 8 and 9 Fundamentals, Lessons 4 and 5
- Java for Beginners, Chapter 3
- Core Java 11 Fundamentals, Lesson 3

4.4. Introduction

Repeating a task multiple times is done using loops in Java. There are three different types of loops:

1. For loops
2. While loops
3. Do-while loops

4.5. For Loops

For loops should be used when we know how many times a task is to be repeated. These are also called as counting loops, as they count the number of times the loop runs. It is useful to know that a for loop can be written as a while loop, but vice-versa is not true.

More details about for loop syntax and examples

4.6. While Loops

While loops are used to repeat actions when we do not know how many times a task is to be repeated. In such cases, we should at least know the signal that indicates when the loop should end. For example: Your instructor asks you to clap your hands until (s)he says "STOP". In this case, you won't know how many times to clap, but you know that the signal to stop clapping is "STOP".

Syntax of the while loop and an example

4.7. Common Loop Algorithms

4.7.1. Java Tutor Example

Java Tutor Example (click “Visualize Execution”)

```
boolean keepLooping = true;
double input;
Scanner keyboard = new Scanner(System.in);
while ( keepLooping ){
  System.out.print("Enter a positive value < 100: ");
  input = keyboard.nextDouble();
  if (0 < input && input < 100){
    keepLooping = false;
  }
}
```

In the above code example, the loop runs as long as the flag (keepLooping) is true and ends when the flag becomes false. The flag becomes false only when the input provided by the user meets the specifications.

Step-by-Step Animation

4.7.2. Java Tutor Example

Java Tutor Example (click “Visualize Execution”)

```
int sum = 0;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter an integer: ");
while (keyboard.hasNextInt()){
   int input = keyboard.nextInt();
   sum = sum + input;
}
System.out.println(“Sum of integers is ” + sum);
```

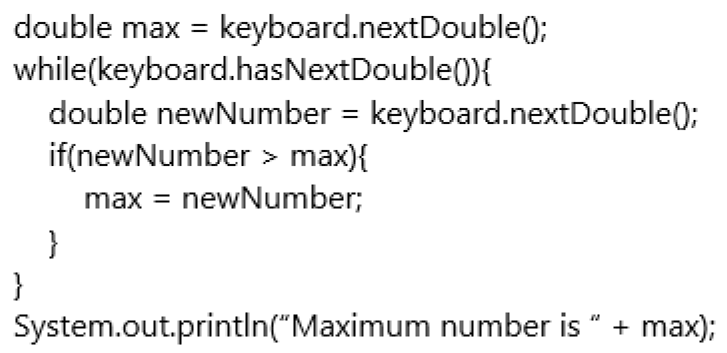
In the above example, the loop uses the hasNextInt() method to check if the next value provided by the user is int. It is important to note that this method does not actually read the value but instead returns true, if the value is int and false otherwise. If the hasNextInt() method returns true, then the nextInt() method reads the value and updates the sum. The loop ends when the user enters a non-integer value to indicate that they do not want to enter any more numbers.

Step-by-Step Animation

This code also demonstrates a common algorithm used to calculate the sum of multiple numbers. hasNextLine(), hasNextDouble() and hasNextBoolean() are other methods that can be used in this strategy depending on the type of input expected.

4.7.3. Java Tutor Example

Java Tutor Example (click “Visualize Execution”)



```
double max = keyboard.nextDouble();
while(keyboard.hasNextDouble()){
    double newNumber = keyboard.nextDouble();
    if(newNumber > max){
        max = newNumber;
    }
}
System.out.println(“Maximum number is “ + max);
```

Step-by-Step Walkthrough

This example uses the hasNext…() method to control the loop. It starts with the assumption that the first value is max. Then compares each new value to the max and updates max if the new value is greater than the current max.

All of the examples shown in this section are examples of user- controlled loops. The user decides when the loop ends.

4.7.4. Sequence generating loops

Loops can be used to generate a sequence of numbers, such as a sequence of the first 10 prime numbers.

More details and examples

4.8. Do-while Loops

Do-while loops are similar to while loops, with one difference. The condition of a do-while loop is checked on exit. This ensures that a do-while loop will run at least once.

More details and examples

A comparison of different types of loops is discussed on lynda.com videos “Comparing Loops” and “Comparing Different Types of Loops”.

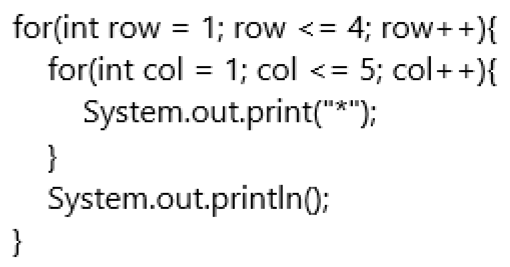
4.9. break and continue

Two keywords, *break* and *continue* can be used to change the flow of a loop in between if needed. The keyword *break* allows to break from a loop and exit it. The keyword *continue* allows to move on to the next iteration.

Details and examples

4.10. Nested Loops

When one loop is placed inside the body of another loop, we have a nested loop. In the following example, a nested loop is used to create a table of "*".



```
for(int row = 1; row <= 4; row++){
   for(int col = 1; col <= 5; col++){
      System.out.print("*");
   }
   System.out.println();
}
```

The outer loop runs 4 times and represents 4 rows of the table. For each value of row, the inner loop runs 5 times, generating 5 columns of "*". Such 5 columns are generated 4 times. Result looks as below.

```
*****
*****
*****
*****
```

As you see in the Step-by-Step Animation, notice both the output and the changing value of variables.

Refer to MathBits for a simple example demonstrating how nested loops work.

This youtube video explains the concept of nested loops.

4.11. Exercises

4.11.1. Exercise 1

Write a program that prompts the user for an integer and displays if the provided integer is a prime number or not. A prime number is a number that is divisible only by 1 and itself. First few prime numbers are 2,3,5,7,11,13 and so on. Sample run is shown below

Sample output for value of 51:
51 is not a prime number

Sample output for value of 83:
83 is a prime number

4.11.2. Exercise 2

Write a program that prompts the user for student grades, calculates and displays the average grade in the class. The user should enter a character to stop providing values.

Sample out for student grades [20, 40, 55, 17, 67, c]:
Average student grade is 39.8

4.11.3. Exercise 3

Write a program that prompts the user for student grades and displays the highest and lowest grades in the class. The user should enter a character to stop providing values.

Sample out for student grades [20, 40, 55, 17, 67, c]:
Highest student grade is 67
Lowest student grade is 17

4.11.4. Exercise 4

Write a program that prints the first 30 values in the Fibonacci series. A Fibonacci series begins with 0 and 1. The next number is then found by adding the previous two numbers. The first few numbers in the Fibonacci series are: 0,1,1,2,3,5,8,13 and so on.

4.11.5. Exercise 5

Write a program that prompts the user for an integer value. The program should then calculate and print the factorial of the user provided value. Factorial of a number, n, written as n! is calculated as a product of all integers less than or equal to n. 5! = 5*4*3*2*1 = 120. 0! = 1. 1! = 1.

4.11.6. Exercise 6

Write a program that accepts an integer from the user and displays the sum of the digits of the provided integer.

```
Sample output for value 235:
Sum of digits of 235 is 10
```

4.11.7. Exercise 7

Write a program that prompts the user for two String values. The program should then display if string 1 is greater in length than string 2. The program should also display if string 1 appears after string 2 in the lexicographic order or vice versa or if they are the same. Lastly, the program should display a sentence created by combining both the string values.

```
Sample output for values "I love" and "GGC":
String "I love" is longer than String "GGC"
String "GGC" appears before String "I love" in lexicographic order
New sentence created is "I love GGC"
```

4.11.8. Exercise 8

Write a program that accepts a String value from the user and displays the reverse of that value.

```
Sample output for value "Hello, World!":
Reverse of "Hello, World!" is "!dlroW ,olleH"
```

For additional challenge, determine if the String and its reverse are equal and display a message explaining the result.

```
Sample output for value "Hello, World!":
String value "Hello, World!" and its reverse "!dlroW ,olleH" are not equal
```

4.11.9. Exercise 9

Write a program that prompts the user for a String value and a character value. The program should then find the last occurrence of the provided character in the provided String and display the corresponding index. If the character is not found in the String, display -1.

```
Sample output for values "Hello, World!" and 'l':
Last occurrence of character 'l' in "Hello World" is at index 10
```

Sample output for values "Hello, World!" and 'g':
Last occurrence of character 'g' in "Hello World" is at index -1

4.11.10. Exercise 10

Write a program that creates the following pattern.

```
******
*****
****
***
**
*
```

4.12. Do You Have Any Questions about Chapter 4?

Comments

5. Methods

5.1. Learning Objectives

1. Define the components of a method header
2. Define and produce a method body
3. Understand parameter passing and use. This will include both pass by value and pass by reference variables.
4. Understand and use class methods.
5. Understand and properly call methods, void and value returning
6. Understand and use instance methods
7. Understand and use the proper return syntax
8. Understand how to use returned values in your calling code

5.2. Resources

5.2.1. Text

- Think Java, Chapters 4 and 6, 6.1 - 6.6: How to Think Like a Computer Scientist, Void Methods and How to Think Like a Computer Scientist, Value Methods by Allen Downey and Chris Mayfield. === Video
- Safari, Deitel Method Video
- Lynda.com: Java Essential Training: Syntax and Structure - Chapter 5. Manage Program Flow, Create reusable code with methods
- Lynda.com: Java Essential Training: Syntax and Structure - Chapter 5. Manage Program Flow, Create overloaded methods
- Lynda.com: Java Essential Training: Syntax and Structure - Chapter 5. Manage Program Flow, Pass arguments by reference vs. value
- Codecademy: Learn Java, Object-Oriented Java - Learn Java: Methods
- YouTube Java Programming 4 - Methods
- YouTube 8.1 Java Tutorial for Beginners: Methods and Functions Part 1
- YouTube 8.2 Java Tutorial for Beginners: Methods and Functions Part 2

5.3. Key Terms

5.3.1. Think Java Vocabulary Chapter 4

- argument: A value that you provide when you invoke a method. This value must have the same type as the corresponding parameter.
- invoke: To cause a method to execute. Also known as "calling" a method.
- parameter: A piece of information that a method requires before it can run. Parameters are variables: they contain values and have types.
- flow of execution: The order in which Java executes methods and statements. It may not necessarily be from top to bottom, left to right.
- parameter passing: The process of assigning an argument value to a parameter variable.
- local variable: A variable declared inside a method. Local variables cannot be accessed from outside their method.
- stack diagram: A graphical representation of the variables belonging to each method. The method calls are "stacked" from top to bottom, in the flow of execution.
- frame: In a stack diagram, a representation of the variables and parameters for a method, along with their current values.
- signature: The first line of a method that defines its name, return type, and parameters.
- Javadoc: A tool that reads Java source code and generates documentation in HTML format.
- documentation: Comments that describe the technical operation of a class or method.

5.3.2. Think Java Vocabulary Chapter 6

- void method: A method that does not return a value.
- value method: A method that returns a value.
- return type: The type of value a method returns.

- return value: The value provided as the result of a method invocation.
- temporary variable: A short-lived variable, often used for debugging.
- dead code: Part of a program that can never be executed, often because it appears after a return statement.
- incremental development: A process for creating programs by writing a few lines at a time, compiling, and testing.
- stub: A placeholder for an incomplete method so that the class will compile.
- scaffolding: Code that is used during program development but is not part of the final version.
- functional decomposition: A process for breaking down a complex computation into simple methods, then composing the methods to perform the computation.
- overload: To define more than one method with the same name but different parameters.

5.4. Overview

Methods are used for several key purposes in programming. First, they allow us to decompose our problems into smaller parts. When we solve complex problems, trying to grasp the entire problem at one time can easily overwhelm our ability to see the solution.

Second, by moving code into methods, we can use this same code in multiple places without having to re-write this code. This allows us to make changes in the way this code executes as requirements change or to correct defects in a single location.

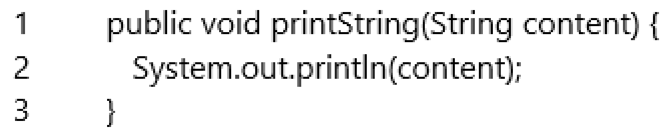
Third, methods allow us to use member variables. A member variable is a variable that is declared in the class and not in a method. This variable will exist throughout the class and can be accessed by all of the methods in the class.

5.5. Method Basics

Methods can either return a value or not return anything. Think Java Chapter 4 discusses void methods, those that don't return a value. Think Java Chapter 6 discusses value returning methods. A method that returns a value can only return a single item, this item can be an object of a class or some type of data structure that contains multiple values.

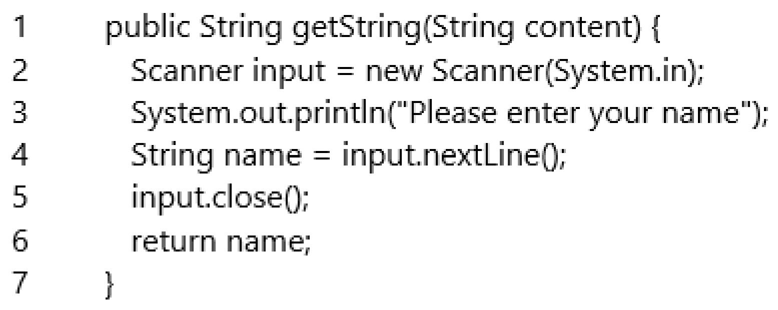
5.6. Example Method

Here is an example method in Java.



```
public void printString(String content) {
  System.out.println(content);
}
```

The following is an example of a value returning method.



```
public String getString(String content) {
  Scanner input = new Scanner(System.in);
  System.out.println("Please enter your name");
  String name = input.nextLine();
  input.close();
  return name;
}
```

5.7. Java Provided Methods

Java provides many methods through the Java API. For example, the Math class, Java API, provides many useful methods including those to provide the absolute value of a number, the maximum of two numbers, the square root of a number, a number raised to the power of another number and so on. Other classes provide methods to read input from the user, Scanner, display output to an output device, PrintStream which we have seen in the System class. The String class provides many built-in methods for processing String data.

5.8. Create Your Own Methods

We can create our own methods to allow us to separate our code into manageable units. Keep in mind that your methods can either be value returning or return nothing. You cannot return more than one item from a method.

5.8.1. Parts of a Method

5.8.2. Method Header

public void printString(String content)

The method header is made up of several key components. First, is the access modifiers. These can be public, private, protected or default. These keywords control what parts of the program have visibility for the method.

- public - the entire program can use this method
- private - only this class can use this method
- protected - only classes in the same package and children of this class can use this method

- default - this is specified by not having a modifier as shown below. This allows items in the same package to use this method.

Note: a package is a container that holds multiple classes, a folder on your disk.

void printString(String content)

The other modifier determines the ownership of the method, either static or not. If we use the keyword static, this method belongs to the class. If we leave this blank, it belongs to an object of the class.

public static void printString(String content)

If we do not use the keyword static, the method belongs to an instance of the class.

public void printString(String content)

Please see Types of Methods below.

The next part of the method header is the return type. Methods must have an explicit return type even if they don't return anything. A non-returning method is declared using the keyword void.

public void printString(String content)

In this example, the method printString does not return anything, notice the keyword void.

public String returnString(String content)

In this example, the method returnString, must return a String.

Next is the method name followed by parenthesis. In the parenthesis are the parameters, either none or as many as you would like to pass to the method. These are declared using the type name convention used by Java. For example, in our example method, we used String content. This tells the method that the first item being passed is of type String and throughout the entire method, its name is content.

public void printString(String content)

In this example, the method name is printString and the parameter passed is content which is of type String.

5.8.3. Parameter Types

5.8.4. Primitive Parameters

A primitive parameter is one of the 8 Java primitive types, byte, short, int, long, float, double, char, boolean. When passing a primitive type parameter, the parameter is passed by copy. This means changes made to it in the method are not reflected in the calling code unless this variable is returned.

5.8.5. Reference Parameters

A reference parameter is any object that we pass to a method. This includes any class type variable, an array or any other type of variable that is initialized using the new keyword. If the object type is mutable, can be changed, changes in the method are reflected in the calling code. String is immutable meaning that changes to a String object delete the original object and instantiate a new String with the changes. For an immutable type, no changes are made in the calling code.

Note: if you have a return type other than void declared in your method, it must return this type or your code will not compile.

5.8.6. Method Signature

The method signature is defined as the name and the types of parameters being passed to it. A method's signature must be unique in a class. Below is the method signature for our example method.

printString(String content)

Note: You cannot use the return type or the modifiers to generate a different method signature.

5.8.7. Parameters

Parameters are the variables that are passed to the method.

public void printString(String content)

In this example, content is the parameter and must be of type String.

Variables declared in the method header are called formal parameters. These exist throughout the method and are used to pass information to the method. A common mistake made by beginning programmers is to reassign these variables in the method rather than use the information passed in. You can pass multiple parameters to a method by separating them with a comma. Remember, order does matter.

Variables in a method are passed in two different ways. First, if the variable is a primitive variable, a copy of the variable is passed to the method. Any changes made to this variable are kept in the method and the variable in the calling code is not changed. If the variable is a reference variable, an object of a class, the address is passed. Since the method has been passed the address of an object, changes made to the object in the method are made to the object in the calling code. Some reference variables are immutable, cannot be changed. These objects, like a String object, are destroyed and re-created when re-assigned. Therefore, it is working on a different object and changes are not reflected in the calling code.

5.8.8. Method Body

The method body contains the code that does the work in the method. Typically, we try to limit this to no more than 30 lines of code. More lines may be an indication that your method should be broken into multiple methods. The Method Body is surrounded by opening and closing curly braces. For example, the method body in our example method is:

```
{
    System.out.println(content);
}
```

5.9. Types of Methods

Methods belong to either the class or an object of the class.

5.9.1. Class methods

A class method belongs to the class and is called using the class name. For example, when we print a line to the screen, we use System.out.println. Since System is a class, the out variab le is static in hte System class. We use the System class name to call the println method in the PrintStream class. This variable belongs to the class and is called using the class name.

When a method is declared using the keyword static in its header. We do not have to instantiate an object of the class to use this method. The line System.out.println tells the JVM to use the System class. In this class is a PrintStream object, this object is declared as static in the System class. The PrintStream object named out contains a method println which writes a line to the standard output stream, in this case, the screen.

5.9.2. JavaTutor Example

Java Tutor Example (click “Visualize Execution”)

To call a class method from within the same class, simply use the method name and associated parameters.

5.9.3. Instance methods

An instance method belongs to an object of the class. It is declared without using the static keyword in the method header. For example, when we use a Scanner to read user input, we must create an object of this class to allow us to use its methods. With the Scanner class, we must instantiate an object to allow us to use this object since a Scanner can be attached to many different objects. When we construct the Scanner using Scanner input = new Scanner(System.in); we are creating a Scanner and attaching it to the default input device, the keyboard. We will learn later that a Scanner can be attached to a file, a String, a socket and other input devices.

5.9.4. JavaTutor Example

Java Tutor example (click "Visualize Execution")

To call an instance method from a class method, main is a class method due to the static modifier, we must create an object of this class and call the method through that object.

5.10. Overloaded Methods

You can have two methods that have the same name as long as they have different numbers or types of parameters. The method name and parameters, the signature, cannot be ambiguous, a method the compiler can confuse with another method. Keep in mind, the signature is the name and the type and number of the variables not the names of the variables. The following two methods are examples of overloaded methods.

5.10.1. JavaTutor Example

Java Tutor example (click "Visualize Execution")

5.11. Calling Methods

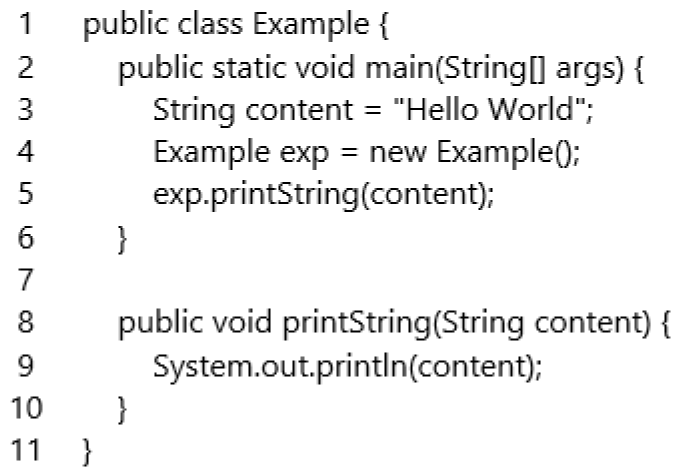
When you call a method, you use the method name and the correct number and type of parameters to call the method. For example, the following code will call our example method.

```
printString("Hello World!!!");
```

If you are calling an instance method from a static context, the main method for example, you must call this method through an object of the class containing the method. For example, the following code will not compile if the printString method is an instance method.

```
public static void main(String[] args) {
   String content = "Hello World";
   printString(content);
}
```

To use this method from the static context, you need to create an object of the class to use to call the method.



```java
public class Example {
   public static void main(String[] args) {
      String content = "Hello World";
      Example exp = new Example();
      exp.printString(content);
   }

   public void printString(String content) {
      System.out.println(content);
   }
}
```

5.12. Member variables

5.12.1. Declaring member variables

Member variables are declared in the class but outside any method. By convention, these are declared at the top of the class to make it easy for programmers to know what they are. A member variable can take one of two forms. First, they can be class variables which means they are declared using the keyword static. This means they are created before your program begins execution in the Java Virtual Machine and that there is only one copy of this variable no matter how many instances of this class are created. The following code creates a class variable of type String named myString. This variable will exist throughout the class and can be accessed by any method in the class. By declaring it private, only the class that contains it can access this variable. Other alternatives include public, every class in your program has access to it; no access modifier which gives it default visibility, every class in the same package (a folder on your computer) can access it; or protected, similar to the default visibility but also allows classes that inherit from this class to access it. Inheritance will be discussed in future classes. The recommend visibility for variables is private. This allows us to encapsulate this information and prevents unwanted modifications.

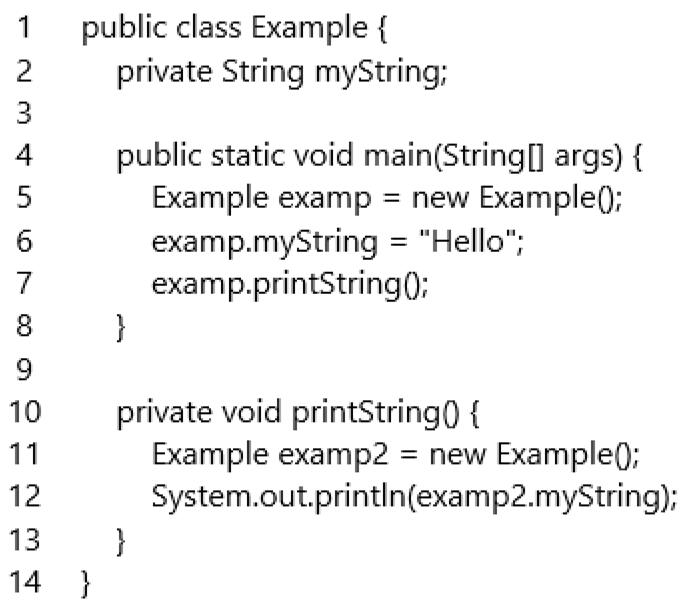
```java
public class Example {
    private static String myString;

    public static void main(String[] args) {
        myString = "Hello";
        printString();
    }

    private static void printString() {
        System.out.println(myString);
    }
}
```

The second type of member variable is an instance variable. This is one that is declared without using the static keyword. Instance variables exist throughout the class but are only created when an object of the class is instantiated. The following code modifies the previous example to be an instance variable.

```java
public class Example {
    private String myString;

    public static void main(String[] args) {
        Example examp = new Example();
        examp.myString = "Hello";
        examp.printString();
    }

    private void printString() {
        System.out.println(myString);
    }
}
```

Notice in the above code, we had to create an instance of the Example class and use that instance to set the value of myString in the main method. You cannot access an instance method from the main method without creating an instance of the object that contains it. This is the same reason we have to create an instance of the Scanner class to access the methods contained in it, these methods are instance methods. In the Math class, we simply call them through the class name, Math.pow(2,3) for example. Since the main method must be declared with the static modifier, it is considered to be a static context. To access an instance variable, we have to tell the main method which instance we want to access.

Caution: if you are working with instance variables and in your method create another instance of your class, you are not working on the same copy of the variable. For example, in the following code, two copies of the Example class are created and when the code is run, the output will be null instead of Hello.



```
public class Example {
   private String myString;

   public static void main(String[] args) {
      Example examp = new Example();
      examp.myString = "Hello";
      examp.printString();
   }

   private void printString() {
      Example examp2 = new Example();
      System.out.println(examp2.myString);
   }
}
```

5.13. Exercises

5.13.1 Exercise 1

Read user input

- Create a method, getString, that allows the user to enter text from the keyboard and return the String entered by the user.
- Note: You can only have a single copy of the no parameter and one parameter methods defined in your class at a time. Start with the class methods and then comment them out when you write the instance methods.

5.13.2 Exercise 2

Read a String (class method, no parameters) Using the keyword static, define this method.

- Create an instance of the Scanner class.
- Prompt the user to enter a String
- Using the Scanner instance, read the String

- Return the String the user entered
- Call the method from the main method

5.13.3 Exercise 3

Read a String (class method, Scanner passed as a parameter) Using the keyword static, define this method.

- Prompt the user to enter a String
- Using the Scanner instance passed to the method, read the String
- Return the String the user entered
- Create an instance of the Scanner class in the main method
- Call the method from the main method passing the Scanner instance as a parameter

5.13.4 Exercise 4

Read a String (instance method, no parameters) Without using the keyword static, define this method.

- Create an instance of the Scanner class.
- Prompt the user to enter a String
- Using the Scanner instance, read the String
- Return the String the user entered
- Create an object of your class in the main method
- Using this object, call your method from the main method

5.13.5 Exercise 5

Read a String (instance method, Scanner passed as a parameter) Using the keyword static, define this method.

- Prompt the user to enter a String
- Using the Scanner instance passed to the method, read the String
- Return the String the user entered
- Create an object of your class in the main method
- Create an object of the Scanner class in the main method
- Using the object of your class, call the method passing the Scanner object as the parameter

5.13.6 Exercise 6

Sum of numbers

- Create a method named sum that takes two numbers and returns the sum of these two numbers.

Sum of Integers

- Create a method sum that takes two parameters, both integers. Do not use the keyword static in this method declaration. This method should return an integer. Create code in your main method that calls this method.

Sum of Floating Point Numbers

- Create a method sum that takes two parameters, both doubles. This method should return a double. Do not use the keyword static. Create code in the main method that calls the sum method with two doubles, with two ints and with one double and one int. Which method gets called in each case. Hint, you may want to put a print statement into each method to help determine which method is called. Why is the specific method called?

5.13.7 Exercise 7

Get User Input

- Write a method, getInput, that allows the user to enter a String and returns this value to be printed using your printString method defined above. Again, do not use the static keyword on your methods other than main.

5.13.8 Exercise 8

Even Number

- Create a class that asks the user to enter a number. Call a method isEven that returns true or false if the number is even. The return from isEven should be passed to printEven which will print "The number is even" if the number is even and "The number is odd" if the number is odd. Determination of what to print must be done based on the return from the isEven method.

5.13.9 Exercise 9

Calculate Fibonacci Sequence

- Write a method, printFib, that takes an integer argument. In this method, create the code required to generate A Fibonacci Sequence with that many numbers. Your main method should contain a loop allowing the user to print multiple sequences, ask them if they want to print another sequence.

5.13.10 Exercise 10

Is Prime

- Write a method, isPrime that takes an int as a parameter and returns true if the number is prime, false if it is not.

5.13.11 Exercise 11

Reverse String

- Create a method reverseString which takes a String as a parameter and returns a String with all of the characters reversed.

5.13.12 Exercise 12

Is Palindrome

- Create a method, isPalindrome, which returns true if the String passed to it is a palindrome and false if it is not.

5.13.13 Exercise 13

Get Address

- Create a class with instance variables to hold the name, street address, city, and state for a user. You should enter the name and address in the nameAddress method. You should enter the city and state in the cityState method. In the main method, print the complete address. You should not use the static keyword except for the main method.

5.13.14 Exercise 14

Get Game Scores

- Create a class that allows users to enter their name and their high score for the game. You should enter the name in a method which returns a String. You should pass the name to a method to allow the user to enter a String. Print the name and score from a method printScore. Allow the user to continue to enter users and scores until they do not enter a name.

5.13.15 Exercise 15

Rectangle size

- Create a method that allows the user to enter the height and width for a rectangle. You will need to use instance variables to hold the values entered. Once you obtain these measurements, call the calculateArea method passing these values to the method. This method should return the area of the rectangle. Once you have the area, call a method isLarge which takes as an int argument containing the area of the rectangle. This method should return true if the area of the rectangle is greater than 300, false if it is less than or equal to 300. Finally, create a method printSize which takes a boolean variable, the return of the isLarge method. If the boolean is true, print "This is a large rectangle." If it is false, print "This is a small rectangle." Create this program using the static keyword only for the main method.

5.14. Do You Have Any Questions about Chapter 5?

Comments

6. Arrays and ArrayLists

6.1. How Does This Topic Relate to Object Oriented Programming?

Arrays and ArrayLists both play an important part in the Object Oriented programming landscape. For both structures, classes with utility methods exist for performing common operations such as sorting and searching. ArrayLists are defined as first class OO objects, with attributes and methods which describe the contents, characteristics and functions of this important class. In a short amount of time, you will frequently be integrating ArrayLists and OO programs in a seamless and very natural manner!

6.2. Learning Objectives

- Learn about a fundamental data structure in programming - the Array
- Learn how to declare, initialize, access and modify arrays
- learn how to process elements in an array through iteration
- Introduce the array's object-oriented construct, the ArrayList
- Learn how to declare, initialize, access and modify ArrayLists
- Understand the tradeoffs of using arrays and ArrayLists
- Learn about advanced capabilities of ArrayLists: such as: adding sublists, removing sublists, search ArrayLists

6.3. Key Terms

Review the important terms in Chapter 8.11 and Chapter 12.11 of ThinkJava.

6.4. Resources

6.4.1. Text

- Think Java Chapter 8 - Arrays
- Think Java Chapter 12 - Arrays of Objects

6.4.2. Video / Tutorial

- Core Java 11 Fundamentals, Second Edition LiveLesson (requires login)
    - Arrays
    - ArrayList Intro Part 1 Part 2 Part 3

6.5. Overview

Until now, we have dealt with variables that deal primarily with a single value. In many programs, it is helpful to treat similar items as a group. The java programming language provides mechanisms to refer to a group with one variable: arrays and ArrayLists.

Why two? That would take a longer explanation, but we can simplify for now by saying that arrays are more efficient and should be used when efficiency is paramount; ArrayLists are less efficient and as a result easier to program and more flexible.

> *Some languages allow programmers to mix data types within an array structure. In general, and although it is possible, such mixing is strongly discouraged when programming in Java.*

6.6. Arrays

6.6.1. Creating Arrays

6.6.2. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.6.3. Indexing Arrays

The index of each value in a list describes its location in the list. Indexes start with the first value at position 0 and end with the last value at position length - 1. The indexes for the list ["Apple", "Plumb", "Kiwi"] are below. Note that the list is of length 3 and thus has indexes from 0 to 2.

Table 7. Indexing

List Values	"Apple"	"Banana"	"Cherry"
Indexes	0	1	2

6.6.4. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

> *Note that since we can access and modify elements in a list, we can also swap them. Can you figure out how to swap the first and last elements of any list?*

6.6.5. Array Properties

6.6.6. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.6.7. Arrays and For Loops

In Java, programmers can process a series of steps multiple times, by looping. Arrays and loops go together well, since it is often desirable to perform some processing on each element in an array. The following examples illustrate looping, first with Java’s *enhanced for loop* and then with an *indexed for loop*.

6.6.8. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.6.9. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.6.10. Arrays Example

Let’s roll two fair, six sided die 5 times and store the results in an array. Additionally, we will tally the number of occurrences of each roll total and print these out with a crude bar chart.

6.6.11. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.7. ArrayLists

As mentioned earlier, ArrayLists are similar to Arrays in their functionality. Also noted previously, ArrayLists are not as efficient as arrays, but are much more flexible.

6.7.1. Creating ArrayLists

6.7.2. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

An astute observer will recognize several subtle nuances in the above code. First, the storage size does not need to be declared at the outset, as with arrays. Next, even though it appears that *odds* will contain primitive *int*s, these are wrapped in objects of type *Integer*.

The observer will also see that several shorthand notation conventions are employed when creating the *evens* ArrayList. The *Integer* parameterized type can be omitted on the right-hand side of the equal sign, whenever the type can be inferred. The literal values (2,4,6) can added to the array directly, taking advantage of a Java technique known as *autoboxing*. (*Autoboxing* converts a primitive to the corresponding wrapper class, when inference is possible. Autounboxing works the same, when conversions are needed in the opposite direction).

In the case of the *special* ArrayList, this code demonstrates that the handling of *double* (/*Double* wrapper class) works similar to the *Integer* examples which proceeded it. Finally, Since *Integer*, *Double* and *String* are all Java classes, ArrayLists operate similarly for the fruits Collection. The similarities extend beyond *String* too, and will apply to any Java Class!

> *It is possible to rewrite the code:*
>
> *ArrayList<Integer> odds = new ArrayList<Integer>();*
>
> *to*
>
> *ArrayList odds = new ArrayList();*
>
> ***However, this is strongly discouraged**. This allowance is made to permit older code to run against newer versions of the Java Virtual Machine (JVM), without modification. The guidance to avoid using this shorthand is provided since coding errors can pass through at compile/build time but will be exposed at a later date, when it is more difficult and costly to correct the issue.*

6.7.3. Indexing ArrayLists

Like with arrays, the index of each value in a list describes its location in the list. Indexes start with the first value at position 0 and end with the last value at position length - 1. The indexes for the *fruits* list are below. Note that the list is of length 3 and thus has indexes from 0 to 2.

Table 8. Indexing

List Values	"Apple"	"Banana"	"Cherry"
Indexes	0	1	2

6.7.4. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.7.5. ArrayList Maintenance - Adding and Removing Elements

6.7.6. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.7.7. ArrayList Methods

ArrayLists have many convenience methods. Popular methods include *length*(), *contains*(), *indexOf*() and *clear*() and *trimTo()*. Documentation for these and others can be found at the Java website.

Additional methods *shuffle*() and *sort*() from the *Collections* class will also work with ArrayLists.

The following example illustrates the use several *ArrayList* and *Collections* methods.

6.7.8. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.7.9. ArrayLists and For Loops

ArrayList traversal is conducted similar to that of array processing. ArrayList values can be visited with a traditional or enhanced for loop syntax.

6.7.10. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.7.11. Dice Rolling Example, Revisited

6.7.12. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

6.7.13. ArrayList to Array Conversion, and Vice-Versa

Sometimes, we will need to convert an array to san ArrayList or vice-versa. The syntax to go back and forth is not very symmetric, since and ArrayList is an object while an array is not. The following code example demonstrates one way to transition back and forth.

6.7.14. JavaTutor Example

Java Tutor example (click “Visualize Execution”)

> *It is important to realize that array < - > ArrayList conversions can be resource-intensive for larger data sets.*

6.8. Exercises

6.8.1. Exercise 1

Create an integer array named dice1 with a size of 10. Populate each array location with a roll of a six-sided die (hint: an int value of 1 through 6). Print the array out using an enhanced for loop.

> *Sample output: dice1 = 1 1 6 2 3 5 1 5 4 5*

6.8.2. Exercise 2

Create an integer array named dice2 with a size of 6. Populate each array location with a roll of a six-sided die (hint: an int value of 1 through 6). Print the array out using an indexed for loop.

> *Sample output: dice2 = 4 5 6 1 4 1*

6.8.3. Exercise 3

Create an ArrayList of Integers named dice3. Generate an Integer representing a roll of a six-sided die 10 times, adding each result to dice3. (hint: generate a random integer value between 1 and 6, inclusive). Print the ArrayList using an enhanced for loop.

```
Sample output: dice3 = 3 5 5 1 2 5 3 2 6 5
```

6.8.4. Exercise 4

Create an ArrayList of Integers named dice4. Generate an Integer representing a roll of a six-sided die 5 times, adding each result to dice4. (hint: generate a random integer value between 1 and 6, inclusive). Print the ArrayList using an enhanced for loop.

```
Sample output: dice4 = 3 2 4 4 1
```

6.8.5. Exercise 5

Consider the following source:

```
int[] list1 = {1, 2, 3, 4, 5, 6, 6, 6, 7, 8, 8, 8, 9, 10};
int[] list2 = {2, 4, 8, 10, 12, 14, 16, 18, 20};
```

Create an new ArrayList named intersection that contains only those items that occur in both lists. If a value is duplicated in either list and it occurs in both lists, it should only occur once in the intersection list. For the lists provided, your ArrayList should contain: 2 4 8 10

6.8.6. Exercise 6

Consider the follow ArrayList:

```
ArrayList<LocalDate> centennials = new ArrayList<>();
centennials.add(LocalDate.of(1776, Month.JULY, 4));
centennials.add(LocalDate.of(1876, Month.JULY, 4));
centennials.add(LocalDate.of(1900, Month.JULY, 4));
centennials.add(LocalDate.of(1976, Month.JULY, 4));
centennials.add(LocalDate.of(2076, Month.JULY, 4));
```

As you can observe, a java programmer has mistakenly entered the 1900 date item into the ArrayList. Without removing the associated centennials.add(...) source line, write the code to remove the errant entry. Print out the resulting ArrayList and size as follows:

```
Before removal:
07/04/1776
07/04/1876
07/04/1900
07/04/1976
```

```
07/04/2076
size = 5

After removal:
07/04/1776
07/04/1876
07/04/1976
07/04/2076
size = 4
```

Hint: you should use the DateTimeFormatter class for formatting.

6.8.7. Exercise 7

Consider the follow ArrayList:

```
ArrayList<LocalDate> centennials = new ArrayList<>();
centennials.add(LocalDate.of(1776, Month.JULY, 4));
centennials.add(LocalDate.of(1876, Month.JULY, 4));
centennials.add(LocalDate.of(1976, Month.JULY, 4));
centennials.add(LocalDate.of(2076, Month.JULY, 4));
```

write the code necessary to determine the ArrayList size.

Sample output: size = 4

6.8.8. Exercise 8

Consider the follow ArrayList:

```
ArrayList<LocalDate> centennials = new ArrayList<>();
centennials.add(LocalDate.of(1776, Month.JULY, 4));
centennials.add(LocalDate.of(1876, Month.JULY, 4));
centennials.add(LocalDate.of(1976, Month.JULY, 4));
centennials.add(LocalDate.of(2076, Month.JULY, 4));
```

write the code necessary to determine if the centennial (1876, at 100 years) is present.

Sample output: centennial present = true

6.9. Do You Have Any Questions about Chapter 6?

Comments

7. Object Oriented Programming

7.1. Learning Objectives

- Be able to differentiate between the data and instructions part of an object definition.
- Select access level/modifiers to achieve appropriate level of encapsulation (public or private only)
- Select appropriate fields to include within a class.
- Design and implement programs where two or more classes interact.
- Understand how packages are used to organize classes and how they are used in import statements.
- Be able to describe the difference between: a class and an object, object reference and primitive variable, object reference and object in memory

7.2. Resources

7.2.1. Text

Think Java, Chapters 11 and 14: How to Think Like a Computer Scientist, Classes and How to Think Like a Computer Scientist, Objects or objects by Allen Downey and Chris Mayfield.

7.2.2. Video

Safari, Deitel Classes and Objects Video

7.3. Key Terms

Chapter 11.10 and Chapter 14.8 of ThinkJava

7.4. Overview

Object oriented programming allows us to program in a style that can result in code that is suitable for large scale software development projects. In this chapter, we will learn how to create a class, how to instantiate an object, the difference between static and non-static methods, visibility specifiers, and explore a simple program that utilizes more than two classes.

7.5. Classes and Objects

Sometimes it is useful to group related data and functions into one class. By combining these two, we can often simplify programming in many ways. This representation of data and functions into one is called a **class**. A running copy of this representation is called an **Object**. The data that is part of an object is called **member variables** and the functions are called **methods**. Member variables are made up of class and instance variables. See Methods to learn about class and instance variables in detail.

7.6. Defining a class

Here is an example of a Class in Java.

```
public class Employee {
   private String name;
   private int rank;

   public Employee(String name, int rank) {
      this.name = name;
      this.rank = rank;
   }

   public int getVacationDays() {
      if( rank == 1 ) {
         return 14;
      } else if ( rank == 2 ) {
         return 10;
      } else {
         return 7;
      }
   }

```

Almost everything in Java is an object except primitives such as short, int, doubles, char, long and boolean. For example, strings and arrays are examples of objects that we have been using frequently in Java. The **new** keyword is used to create an object in Java. When creating a class, it is important to make sure that the contents of a class show high **cohesion**. For example, in the

Employee class, all the member variables and methods should be about employees. If a method such as **sendChatMessage()** which sends a chat message to a coworker would be out of place an lessen the cohesion of the class.

7.7. Constructor

```
Employee bob = new Employee("Bob",1);
```

When you create an object using the **new** keyword, a special method called the **constructor** is executed. The constructor is used to initialize instance variables and prepare the object which is created in a special space in memory called the **heap**. The constructor's name and the class name have to match. If a constructor is not present in the class, class, a default constructor is provided instead. The default constructor has no parameters and initializes member variables as follows: numeric primitives to 0, booleans to false, char to '/u0000' and reference variables to null. When any constructor is created, the default constructor no longer exists. Constructors with different method signatures (constructor overloading) are allowed.

The following is the no argument constructor:

```
public Employee() {

}
```

The no argument constructor could be used to set default values such as the following:

```
public Employee() {
    name = "Bob";
    rank = 1;
}
```

The default constructor does not initialize member variables in a meaningful way so a constructor with parameters can be useful.

```
public Employee(String name, int rank) {
    this.name = name;
    this.rank = rank;
}
```

In the above constructor, we see the **this** keyword for the first time. The **this** keyword is a reference to the current object and it can be used to refer to the instance variables of the current object. The **this** keyword can also be used to call other constructors in the following manner.

```java
public Employee() {
    this("Bob",1);
}
```

For more information about the **this** keyword, you can read more about it in the Java Tutorial

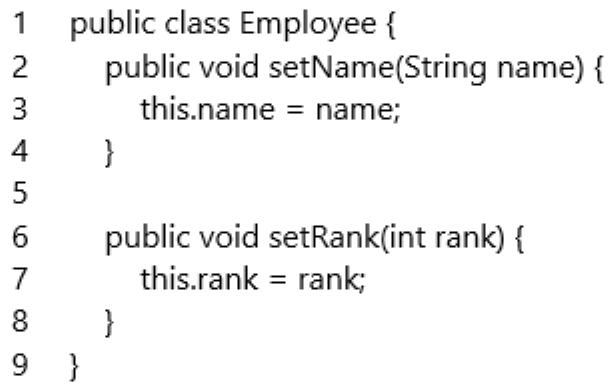
7.8. Getters and Setters

Information hiding refers to the idea that member variables of a class should be closed for modification from outside of class to prevent unintentional or intentional modification and prevent complexity arising from such unexpected modifications. In Java, the **public**, **private**, and **protected** keywords are used to control access to member variables and methods of a class. You can refer the Java tutorial to learn the details but the **public** keyword indicates that you can access the variable or method from anywhere and the **private** keyword indicates that you can access them from only the class.

You can use getters to access private member variables. Getters refer to methods that return the value of private member variables and the convention is to name such methods with the prefix *get*. Here is an example from Employee class.

```java
public class Employee {
    public String getName() {
        return name;
    }

    public int getRank() {
        return rank;
    }
}
```

Setters are methods that are used to change the value of private member variables. The convention is to name setters with the prefix *set*. Here is an example from the Employee class.



```
public class Employee {
   public void setName(String name) {
      this.name = name;
   }

   public void setRank(int rank) {
      this.rank = rank;
   }
}
```

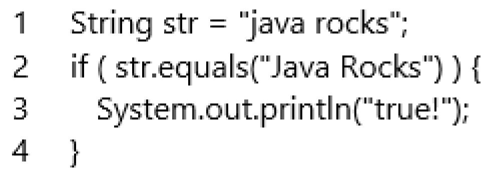
7.9. Static vs. Instance (non-static) methods

Instance methods are methods that belong to each object but static methods are methods that belong to a class. For example, the **sort()** method in the Collection class is an example of a static method. Utility methods such as the **sort()** method do not need to be included with each object so they are part of a class. In fact, the **main()** method is an example of a static method.

In object-oriented programming, it is important not to overuse static methods. When static methods are overused, the programming style can resemble the procedural programming style more than the object-oriented style. At the beginning of your journey in object-oriented programming, try to minimize the creation and use of static methods in your classes. There is also a memory footprint problem with the overuse of static variables and static methods. This will be covered in ITEC 2150 when you learn about garbage collection. Please see the Methods chapter for more information about static and instance methods.

7.10. Implementing the *equals()* and *toString()* method

When using a String in Java, it was necessary to use the **equals** method to establish the equality of two objects.



```
String str = "java rocks";
if ( str.equals("Java Rocks") ) {
   System.out.println("true!");
}
```

The reason is because strings in Java are objects and you cannot use the == operator to test equality. To use the equals() method to test equality, it is necessary to implement the **equals()**

method. The following is equals() method for the Employee class shows that two Employee objects are equal if the name is equal (this is somewhat unrealistic in the real world).

```
public boolean equals(Object obj) {
   if (obj instanceof Employee) {
      return name.equals((Employee)obj.getName());
   } else {
      return false;
   }
}
```

Sometimes it is necessary to get the string representation of an object. In Java, the **toString()** method is used to get the string representation of an object. For the Employee class, we can represent the Employee object with the **name** and **rank** properties.

```
public String toString() {
   return name + " " + rank;
}
```

If you implement the **toString()** method, you can use it in the **System.out.println()** method to print the string representation of the object.

```
Employee e = new Employee("Bob",1);
System.out.println(e);
```

7.11. Using two or more classes together

Suppose you have a **Leadership** class which holds an ArrayList of employees that are managers.

```java
public class Leadership {
   private ArrayList<Employee> managers;

   public Leadership() {
      managers = new ArrayList<Employee>();
   }

   public void addManager(Employee e) {
      managers.add(e);
   }
}
```

You can see that an object (ArrayList of employees) can be part of another object (Leadership). In object-oriented programming, we call this relationship a **has-a** relationship. Object-oriented programming focuses on such relationships among objects and the communication among them.

7.12. Exercises

7.12.1. Create the Student Class

Create a simple class named **student** with the following properties:

- id
- age
- gpa
- credit hours accomplished

Also, create the following methods:

- Constructors
- Getters and setters

7.12.2. Create the *equals()* and *toString()* method for the Student Class

Two students objects are considered equal if their *id* is the same. The toString() method should print out the name and id of the object.

7.12.3. Create the School Class

Create a class called **School** that holds an ArrayList of students. Create the following methods for the class.

- Constructor
- void addStudent(Student)
- void removeStudent(Student)
- Student findYoungestStudent()
- Student findOldestStudent()

7.13. Do You Have Any Questions about Chapter 7?

Comments