```
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.

```
1    public void printString(String content) {
2      System.out.println(content);
3    }
```

The following is an example of a value returning method.

```
1       public String getString(String content) {
2         Scanner input = new Scanner(System.in);
3         System.out.println("Please enter your name");
4         String name = input.nextLine();
5         input.close();
6         return name;
7       }
```

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:

```
1       {
2           System.out.println(content);
3       }
```

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.

```
1    public static void main(String[] args) {
2        String content = "Hello World";
3        printString(content);
4    }
```

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

```
1   public class Example {
2       public static void main(String[] args) {
3           String content = "Hello World";
4           Example exp = new Example();
5           exp.printString(content);
6       }
7
8       public void printString(String content) {
9           System.out.println(content);
10      }
11  }
```

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.

```
1   public class Example {
2       private static String myString;
3
4       public static void main(String[] args) {
5           myString = "Hello";
6           printString();
7       }
8
9       private static void printString() {
10          System.out.println(myString);
11      }
12  }
```

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.

```
1   public class Example {
2       private String myString;
3
4       public static void main(String[] args) {
5           Example examp = new Example();
6           examp.myString = "Hello";
7           examp.printString();
8       }
9
10      private void printString() {
11          System.out.println(myString);
12      }
13  }
```

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.

```
1   public class Example {
2       private String myString;
3
4       public static void main(String[] args) {
5           Example examp = new Example();
6           examp.myString = "Hello";
7           examp.printString();
8       }
9
10      private void printString() {
11          Example examp2 = new Example();
12          System.out.println(examp2.myString);
13      }
14  }
```

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.

```
1   public class Employee {
2      private String name;
3      private int rank;
4
5      public Employee(String name, int rank) {
6         this.name = name;
7         this.rank = rank;
8      }
9
10     public int getVacationDays() {
11        if( rank == 1 ) {
12           return 14;
13        } else if ( rank == 2 ) {
14           return 10;
15        } else {
16           return 7;
17        }
18     }
19
```

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:

```
1   public Employee() {
2
3   }
```

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

```
1   public Employee() {
2       name = "Bob";
3       rank = 1;
4   }
```

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

```
1   public Employee(String name, int rank) {
2       this.name = name;
3       this.rank = rank;
4   }
```

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.

```
1   public Employee() {
2       this("Bob",1);
3   }
```

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.

```
1   public class Employee {
2       public String getName() {
3           return name;
4       }
5
6       public int getRank() {
7           return rank;
8       }
9   }
```

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.

```
1  public class Employee {
2     public void setName(String name) {
3        this.name = name;
4     }
5
6     public void setRank(int rank) {
7        this.rank = rank;
8     }
9  }
```

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.

```
1  String str = "java rocks";
2  if ( str.equals("Java Rocks") ) {
3     System.out.println("true!");
4  }
```

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).

```
1   public boolean equals(Object obj) {
2       if (obj instanceof Employee) {
3           return name.equals((Employee)obj.getName());
4       } else {
5           return false;
6       }
7   }
```

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.

```
1   public String toString() {
2       return name + " " + rank;
3   }
```

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

```
1   Employee e = new Employee("Bob",1);
2   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.

```
1   public class Leadership {
2       private ArrayList<Employee> managers;
3
4       public Leadership() {
5           managers = new ArrayList<Employee>();
6       }
7
8       public void addManager(Employee e) {
9           managers.add(e);
10      }
11  }
```

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