Java Chapter 5 Part 1

* Introduction to Methods: void Methods, Arguments
* CIS 255 • Shelby-Hoover Campus

Programmer-Defined Methods

* In the programs examined thus far, all statements have been written inside the main method
  + This is not a problem for short programs
  + Many complex programs involve thousands or millions of lines of code
  + Placing a large amount of code in one method makes the program difficult to follow
* You’ve already used other methods built into the Java language
* You may want to break your own programs into separate methods

Advantages of Methods

* A programmer finds it easier to focus on one subset of a program’s functionality at a time (**divide and conquer**)
* The code in a method may be executed several times in a program without having to re-enter the code itself (**code reuse**)
* A well-written method that provides common functionality may be reused in other programs
* Different programmers or teams of programmers may work on different methods at the same time
* A method may be tested in isolation to ensure that it works well before it is integrated into the program

Classifying Methods

* The methods covered in chapter 5 are not object-oriented: **static methods** can be used without declaring an object of a class type
  + The keyword static in the method header indicates this difference
  + Without this keyword, Java expects an object
* A method may **return** a value back to the calling method
  + **void methods** handle all output themselves and do not send values back to the calling method (e.g., showMessageDialog, print, etc.)
  + **value-returning methods** allow the calling method to use a calculated value as desired (e.g., showInputDialog, parseDouble, etc.)

Defining a Method: Header

* A method **definition** specifies the statements it contains and how these statements are executed
* Each definition is inside the class definition brackets, but not inside another method definition
* The **method header** is the first line of the definition and must include the following:
  + A modifier indicating whether the method can be used in other programs (public) or only in this program (private)
  + If an object is not required, the keyword static
  + A return type (void or, as shown in the next lecture, a data type)
  + The method’s name (must be unique – no variable should have the same name – and must follow Java rules for identifiers)
  + A set of parentheses (empty for now)
  + No semicolon at the end!

Defining a void Method: Body

* The **method body** (the set of statements to be executed) is enclosed in a set of curly brackets
* Any variables to be used in this method must be declared (each method has its own set)
* void methods are not required to contain any specific statements, but calculated values should be sent to output (the console, dialogs, or a file) within the method

void Method Examples

* Displaying a simple message:  
    
  public static void displayMessage()  
  {  
   System.out.println("Howdy!");  
  }
* Displaying the value of a local variable:  
    
  public static void displayValue()  
  {  
   int a = 3;  
   System.out.println("a = " + a);  
  }

Calling a void Method

* A program’s execution always begins with main regardless of the order of the methods inside the class brackets
* To indicate when another method’s statements should be executed, you need to place a **call** to that method inside the current method
* The call statement for a void method includes the method name, a set of parentheses, and a semicolon:  
    
  displayMessage();
* After the execution of a method is complete, program execution continues with the next statement in the method that contains the call
* Example: SimpleMethod.java (Code Listing 5-1)

Nested Method Calls

* If a method call is placed within a loop body, that method will be called once during each iteration (LoopCall.java, Code Listing 5-2):  
    
  for (int i = 0; i < 5; i++)  
   displayMessage();
* A method call inside a decision structure will only execute if the value of the Boolean expression leads to that particular outcome (CreditCard.java, Code Listing 5-3):  
    
  if ((salary >= 25000) && (creditScore > 670))  
   qualify();  
  else  
   noQualify();

Hierarchical Method Calls

* Any method can call another method
* When a method terminates, control of the program returns to the method that contains the call (not necessarily main)
* When returning from a void method, the program executes the next statement after the method call
* Example: DeepAndDeeper.java (Code Listing 5-4)

Introduction to Arguments

* Many of the predefined methods you have used so far have required **arguments** to provide the method with initial data
  + System.out methods: the text to be displayed
  + Math methods: the operands for the calculation
  + Each method has rules regarding what type of, and how many, arguments are required in the call
* In order for a method to receive arguments, there must be a **parameter list** inside the parentheses of the method header

Writing a Parameter List

* Place the parameters inside the parentheses after the method name
* Each parameter must have a type and a name (you can’t list the type one time and assume it will apply to every parameter)
* Separate the parameters with commas
* Order the parameters as you expect the arguments to be ordered in the method call
* In the method body, use the parameter names to refer to the arguments received

Parameter List Example

* Determining eligibility for a tax credit based on annual income and marital status:  
    
  public static void showEligibility(double income, char married)  
  {  
   if ((income <= 75000) && (married == 'N'))  
   {  
   System.out.println("Eligible.");  
   }  
   else if ((income <= 150000) && (married == 'Y'))  
   {  
   System.out.println("Eligible.");  
   }  
   else  
   {  
   System.out.println("Not eligible.");  
   }  
  }

Writing an Argument List

* Arguments are assigned to the parameters in the order in which they are listed in the parentheses of the method call
* Do not include data types with arguments
* Separate multiple arguments with commas
* An argument may be a literal value, a variable, or an expression that matches the parameter’s type or an equivalent
  + Widening conversions are allowed (e.g., an int argument to a double parameter)
  + Narrowing conversions require a cast
  + The argument variable name may be the same as, or different than, the parameter name (independent)

Argument List Examples

* With variables:  
    
  double salary;  
  char marital;  
  String inputLine;  
    
  Scanner keyboard = new Scanner(System.in);  
    
  System.out.print("Enter your salary: $");  
  salary = keyboard.nextDouble();  
    
  System.out.print("Are you married? (Y / N)");  
  keyboard.nextLine(); // clears buffer  
  inputLine = keyboard.nextLine();  
  marital = inputLine.charAt(0);  
    
  showEligibility(salary, marital);
* With literal values:  
    
  showEligibility(90000, 'N');
* With expressions:  
    
  showEligibility(baseSalary + raise, 'Y');
* Another example of a method with parameters and various calls with arguments: PassArg.java (Code Listing 5-5)

Parameters and Other Variables

* Parameter variables are local variables
* Don’t declare another variable with the same name inside the body of the same method (unless it’s within a separate scope)
* Parameter variables and other local variables in the same method may be used together
* Each method’s variables are in separate memory locations
  + Variables in different methods can have the same name, but their values are separate and unrelated (LocalVars.java, Code Listing 5-8)
  + One method does not have access to another method’s local variables
  + You must use arguments and parameters to send values from one method to another

Primitive and Object Arguments

* Primitive arguments are **passed by value** to a method’s parameters
  + The parameter is assigned a copy of the argument
  + Changes to the parameter variable only affect the parameter within the method during the method’s execution
  + The argument remains unchanged after the method terminates
  + Example: PassByValue.java (Code Listing 5-6)
* Objects are **passed by reference** to a method’s parameters of a class type
  + The parameter uses the same address as the argument
  + Changes to the parameter variable could affect the argument in the original method
  + Some objects (e.g., String objects) are **immutable**
    - An existing String object cannot be changed
    - If a String parameter is assigned a new value, a separate object is created; the argument object remains as-is
    - Example: PassString.java (Code Listing 5-7)

Method Documentation

* Many programmers include a javadoc comment before the header for each method to describe its purpose
  + A javadoc comment begins with /\*\*
  + It may span as many lines as needed
  + It ends with \*/
* The first line of a method’s javadoc comment describes the method’s purpose
* A programmer may include a description of each parameter beginning with the tag @param and the parameter name
* Example: TwoArgs2.java (end of section 5.2)
* In the next lecture, you’ll see another tag used to document a value-returning method