Lecture Nine – Methods and Classes

- Introduction
- SimpleCalc2 program
- Methods
- Return Values
- Passing Values
- Student program
- Adding Classes
- static keyword
- Extending Classes
- Method Overloading
- Drill

Read Chapter 7
Introduction

- So far we have discussed how to write programs in Java and have been skipping some of the details of why things work the way they do.

- You may have found yourself asking questions like, “What do all the parts of `public static void main(String args[])` mean?” and “What is the difference between a class and a method?”

- This lesson is meant to clarify some of these details and to introduce the concepts of methods and classes.
SimpleCalc2 program

- Back in Lesson 6 we wrote our first GUI: SimpleCalc

- This program (like most of our programs so far) was very structured and straightforward, not using many methods (or functions)

- For this program we will re-write our SimpleCalc program but we will use methods instead

- As always, write the code and get it to work first and then we will discuss it in detail
SimpleCalc2 program

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class SimpleCalc2 extends JFrame implements ActionListener, WindowListener
{
    private JLabel enterALabel;
    private JTextField AField;
    private JLabel enterBLabel;
    private JTextField BField;
    private JLabel sumLabel;
    private JTextField sumField;
    private JButton addButton;
    private JButton subButton;
    private JButton mulButton;
    private JButton divButton;
}
public static void main(String args[]) {
    JFrame frame = new SimpleCalc2();
    frame.setSize(250, 150);
    frame.setTitle("Simple Calculator 2");
    frame.setVisible(true);
}

public SimpleCalc2() {
    enterALabel = new JLabel("Enter first value: a = ");
    AField = new JTextField("0", 6);
    enterBLabel = new JLabel("Enter second value: b = ");
    BField = new JTextField("0", 6);
    sumLabel = new JLabel(" The Sum is a + b = ");
    sumField = new JTextField("0", 6);
    addButton = new JButton("Add");
}
FlowLayout layout = new FlowLayout();
Container window = getContentPane();
window.setLayout(layout);

window.add(enterALabel);
window.add(AField);
window.add(enterBLabel);
window.add(BField);
window.add(sumLabel);
window.add(sumField);
window.add(addButton);

addButton.addActionListener(this);
addWindowListener(this);
}
SimpleCalc2 program

```java
public void actionPerformed(ActionEvent e) {
    String tempString;
    int a, b, sum;
    if (e.getSource() == addButton) {
        tempString = AField.getText().trim();
        a = (new Integer(tempString)).intValue();
        tempString = BField.getText().trim();
        b = (new Integer(tempString)).intValue();
        sum = add(a, b);
        sumField.setText(tempString.valueOf(sum));
    }
}
```
SimpleCalc2 program

public void windowClosing(WindowEvent e) {
    System.exit(0);
}

public void windowActivated(WindowEvent e){}

public void windowClosed(WindowEvent e){}

public void windowDeactivated(WindowEvent e){}

public void windowIconified(WindowEvent e){}

public void windowDeiconified(WindowEvent e){}

public void windowOpened(WindowEvent e){}

public static int add(int num1, int num2) {
    int sum;
    sum = num1 + num2;
    return sum;
}
}
Most of this program is exactly the same as our first SimpleCalc program.

The only difference is that we created a method called `add()` that will add two numbers (NOTE: when you see a name with an empty set of parentheses following it we are referring to the name of a method – regardless of whether or not the method actually contains an empty parameter list).

We use methods to help us break code up into logical units that can be re-used and easily re-written. For example, just think if you had to write the same 10 or so lines of code to achieve a certain task. Now imagine that these 10 lines of code appear 20 different times in your program. Now imagine that you need to change the operation of that task. Wouldn’t it be easier to change those 10 lines of code once instead of 20 times?!

There are two parts in the process of using methods:
1. Write the method
2. Call the method
Methods

- In our program we wrote the `add()` method at the end of our class
  ```java
  public static int add(int num1, int num2) {
      int sum;
      sum = num1 + num2;
      return sum;
  }
  ```

- We then called our `add()` method in the `actionPerformed` method
  ```java
  sum = add(a, b);
  ```

- Let’s take a look at each part of the method we wrote
Methods

```java
public static int add(int num1, int num2) {
    int sum;
    sum = num1 + num2;
    return sum;
}
```

- **public** - This first keyword defines the access level of the method. By tagging the method as **public** we are saying that any method in any other class can call it. We will discuss other access levels in the next lesson.
public **static** int add(int num1, int num2) 
{
    int sum;
    sum = num1 + num2;
    return sum;
}

- **static** – This keyword is used to make the method a class method. Class methods are methods that do not operate on objects (more on this later). It is important to note that static methods can ONLY access other static methods and variables. Since our main method is declared as static, we must declare our method as static so it can be accessed by main.
Return Values

```java
public static int add(int num1, int num2) {
    int sum;
    sum = num1 + num2;
    return sum;
}
```

- **int** – This is used to define the method’s return type. Methods are capable of returning a return value to the calling method (in our example the calling method would be `main`). The return keyword is used to specify which value needs to be returned to the calling method. If a method does not return a value then the keyword `void` is used as the return type.
Passing Values

public static int add(int num1, int num2) {
    int sum;
    sum = num1 + num2;
    return sum;
}

- add(int num1, int num2) – The final part of our method header is the name of the method (add) followed by an argument list that is enclosed by parentheses. Adding a non-empty argument list to our method’s allows us to pass values to it. Each argument (or parameter) is specified with a type followed by a name, and each argument is separated by a comma.
Passing Values

```java
sum = add(a, b);
```

- This is the line of code we used in our `actionPerformed()` method that was used to call our `add()` method.
- It is very important to pay attention to the **type** and **order** of your arguments when calling a method.
- Since both arguments were declared to be of type `int` in the header of our `add()` method then that means both `a` and `b` must be of type `int`.
- Order matters as well. In this example the value of `a` will be passed in for the `num1` variable in our `add()` method and the value of `b` will be passed in for the `num2` variable.
Student program

- We now know how we can use methods to help break up our actions and tasks into smaller modules.

- Likewise, we can use multiple classes to help us break our program up into smaller objects.

- Object-oriented programming will be the subject of our next lesson so we will discuss the importance of objects next time.

- Let’s look at an example though – in this program we want to create two “objects”: a Person and a Student. The purpose of the program is simply to output a student’s name and grade. We will not use a GUI for this example (simply for space reasons). **NOTE:** even though this program contains 2 classes this is all one file that should be named “Student.java”
import java.lang.Math;

// Our Person class
class Person {
    static String first;  // first name
    static String last;   // last name
    static int age;       // age

    // method used to return the Person’s full name
    public static String fullname() {
        return first + " " + last;
    }
}

// Our Student class
public class Student extends Person
{
    static float grade;       // grade for student

    public static void main(String[] args)
    {
        int roundedGrade;
        char letterGrade;
        char roundedLetterGrade;

        first = "John";          // set name, age, and grade for Student
        last = "Doe";
        age = 18;
        grade = 89.5f;
Student program

// calculate rounded grade
roundedGrade = Math.round(grade);

// call getLetterGrade() method with a float argument
letterGrade = getLetterGrade(grade);

// call getLetterGrade() method with an int argument
roundedLetterGrade =
    getLetterGrade(roundedGrade);

// output Student’s fullname, grade, and rounded grade
// NOTE: fullname() is inherited from Person
System.out.println(fullname() + " has a grade of " +
    grade + " which is a " + letterGrade +
    " and rounded is a " + roundedLetterGrade);
public static char getLetterGrade(float grade) {
    char letterGrade;
    if (grade >= 90.0)
        letterGrade = 'A';
    else if (grade >= 80.0)
        letterGrade = 'B';
    else if (grade >= 70.0)
        letterGrade = 'C';
    else if (grade >= 60.0)
        letterGrade = 'D';
    else
        letterGrade = 'F';
    System.out.println("Calculated grade with a float argument...");
    return letterGrade;
}
// getLetterGrade() method with int argument
public static char getLetterGrade(int grade) {
    char letterGrade;
    if (grade >= 90)
        letterGrade = 'A';
    else if (grade >= 80)
        letterGrade = 'B';
    else if (grade >= 70)
        letterGrade = 'C';
    else if (grade >= 60)
        letterGrade = 'D';
    else
        letterGrade = 'F';
    System.out.println("Calculated grade with an int argument...");
    return letterGrade;
}
Adding Classes

- This example illustrates how we can have two classes in one program.

- Furthermore, we were able to put both classes in a single file. We were able to do this because our Student class was the only class with the public keyword appearing in front of it. It is the class that the file is named after and also contains the main method.

- However, it is more common to separate your classes into separate files (we will look at an example of this in the next lesson). In our example, we could have accomplished this by declaring the class Person to be public and putting it into a file named Person.java
static keyword

- A class is typically made up of a series of member variables and member methods.

- Our first class, **Person**, contains 3 member variables (**first**, **last**, **age**) and a single member method (**fullname()**).

- The **static** keyword is used in all of our variables to make them **class variables**. This allows them to be accessed throughout the class and in other **static methods**.

- In addition, the **static** keyword also means that all instances of the class will share that same variable instead of getting their own copy of the variable (but we will not worry too much about this right now).
Extending Classes

- You may have noticed that when we declared our `Student` class we added `extends Person` to the end of the declaration.

- This process of extending classes is also known as inheritance.

- In our example, `Person` is called the superclass (the class inherited from) and `Student` is called the subclass (the class that inherits from the superclass).

- When a class extends another class it receives all of the member variables and member methods that are in the superclass.
Extending Classes

- In our example the class **Student** extends **Person**

- Therefore, in addition to having the member variable **grade** and the two **getLetterGrade()** member methods, **Student** also inherits the member variables **first**, **last**, and **age** and the member method **fullname()** from **Person**

- **Inheritance** allows for re-usability of objects. Since a student is a person, it makes logical sense to inherit its members. Likewise, if we decided to create a new class named **Faculty** it would also make sense to re-use the **Person** class and inherit from it.

- **Inheritance** can extend further than a single level. For example we could have a class named **Freshman** than extends **Student** (which extends **Person**). In this case, it would contain its own members along with the members from both the **Student** and **Person** classes.
Method Overloading

- A final observation that you may have noticed is that we are using two member methods in our Student class that have the same name (getLetterGrade()) but a different argument list.

- This is an example of method overloading.

- Although in our example both methods essentially do the same thing, there may be times when we want to do something different in our method depending on the types (and number) of arguments passed into the method.
Drill 9

- Modify **SimpleCalc2** to do subtraction, division, and multiplication (much like the Drill from Lesson 6) but use methods like we illustrated in this lesson.

- Create a class called **Account** that contains member variables `accountNo`, `customerName`, and `balance`. Add member methods `deposit()` and `withdraw()` that take float arguments and either add (or subtract) the amount passed as an argument to the `balance`. Overload these methods to also allow int values to be passed in. Then, create a subclass of **Account** called **SavingsAccount** that has a member variable called `interestRate`. Create a member method for **SavingsAccount** called `addInterest()` that adds interest to the balance on the account. For example, if our balance is $100 and our interest rate is 5% then `addInterest()` will add $5 to our balance, making it $105. In your main method initialize all of the member variables for the **SavingsAccount**. Make a few deposits, a few withdrawals, and add in interest. Finally, display the account number, the customer’s name, the interest rate of the account, and the final balance.