Unit 5: Writing Classes
Writing Methods

Adapted from:

1) Building Java Programs: A Back to Basics Approach

by Stuart Reges and Marty Stepp

2) Runestone CSAwesome Curriculum

Textbook Reference

Online Textbook Think Java - 2nd Edition by Allen Downey and Chris Mayfield

For this lecture use Chapter 9 10 11

Modularity

modularity: Writing code in smaller, more manageable components or modules. Then combining the modules into a cohesive system.

– Modularity with methods. Break complex code into smaller tasks and organize it using **methods**.

Methods define the behaviors or functions for objects.

An object's behavior refers to what the object can do (or what can be done to it). **A method is simply a named group of statements.**

static vs non-static

Variables and methods can be classified as **static** or **nonstatic(instance)**.

Non-static or instance: Part of an object, rather than shared by the class. Non-static methods are called using the dot operator along with the object variable name.

static: Part of a class, rather than part of an object. Not copied into each object; shared by all objects of that class. Static methods are called using the dot operator along with the class name unless they are defined in the enclosing class.

Instance Methods

Non-static or instance methods belong to individual objects. They are usually implemented inside of an object class rather than the driver class.

Methods in an object class are non-static or instance by default unless explicitly labeled "static".

Non-static methods are called through objects of the class.

```
public type name(parameters) {
          statements;
    }
```

Static methods

static method: Stored in a class, not in an object.

Shared by all objects of the class, not replicated.

Does not have any **implicit parameter**, `this`; therefore, cannot access any particular object's instance variables.

```
public static type name(parameters) {
    statements;
}
```

Method Returns

Methods in Java can have **return types**. Such **non-void** methods return values back that can be used by the program. A method can use the keyword "**return**" to return a value.

```
public type methodName(type var1,…, type var2){

…

}
```
Examples:
```
public int method1(){

…

}



public double method2(int x){

…

}
```
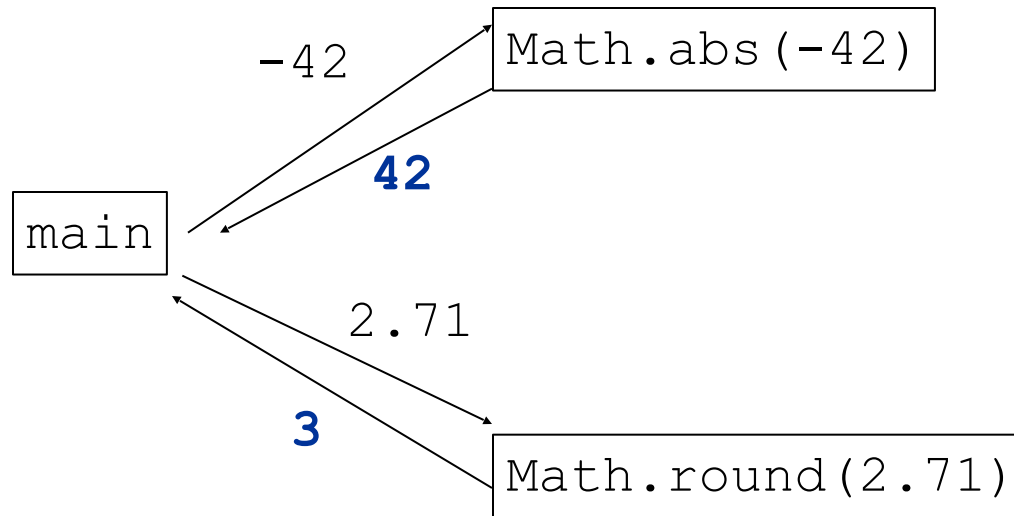
return types

Note: Method parameters are its inputs and method returns are its outputs.

Return

- **return**: To send out a value as the result of a method.
  - The opposite of a parameter:
    - Parameters send information in from the caller to the method.
    - Return values send information out from a method to its caller.
      - A call to the method can be used as part of an expression.

Calling Non-void Methods

A non-void method does return a value and should be stored or printed. Otherwise, that value will be lost.

```java
public class MyClass{
      public static void main(String[] args){
            int a = 3 + printX(5); //error! Does not return!
            int b = twiceX(3); // correct, b = 6
            twiceX(10); // value is lost, this does nothing.
      }
      public static void printX(int x){
            System.out.println("The input x is" + x);
      }
      public static int twiceX(int x){
            return 2 * x;
      }
}
```
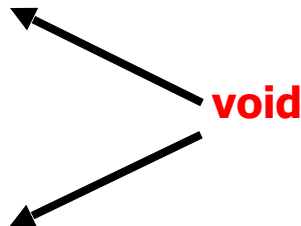
Void Methods

Void methods do not have return values. Once the execution of the method completes, the flow of control returns to the point immediately following where the method was called.

```
public void methodName(type var1,…, type var2){
…
}
```
Examples:
```
public void method1(){
…
}                           void


public void method2(int x){
…
}
```

Calling Void Methods

Code from a method can directly call another method in the same enclosing class without using the dot notation.

Void methods do not have return values and are therefore not called as part of an expression.

```java
public class MyClass{
    public static void main(String[] args){
        printX(5); // Output: The input x is 5
    }
    public static void printX(int x){
        System.out.println("The input x is" + x);
    }
}
```

Calling Methods with Parameters

When calling a method with parameters, values provided in the parameter list need to correspond to the order and type in the method signature.

```java
public class MyProgram{
  public static void main(String[] args){
        double ave = average(4, 5); // saved return value
        System.out.println(sum(2.4, 5.1)); // print it
        sum(4.0, 3.2);     // returned value is lost
                           // DON"T DO THIS!
  }
  public static double average(int x, int y){
  …
  }
  public static double sum(double x, double y){
  …
  }
}
```

Calling Instance Methods

Use the dot notation when calling a method in not in the enclosing class. When calling an instance method, use: objectName.methodName(parameters);

MyProgram.java

```
public class MyProgram{
   public static void main(String[] args){
        Student a = new Student();
        a.setName("Michele Li");
        System.out.println(a.getName());
   }
}
```

MyClass.java

```
public class Student{
   …
   public String getName(){
    …
   }
   public void setName(String new_name){
    …
   }
}
```

Calling Non-instance Methods

Use the dot notation when calling a method in not in the enclosing class. When calling static or non-instance method, use: className.methodName(parameters);

MyProgram.java

```
public class MyProgram{
  public static void main(String[] args){
        Student a = new Student();
        a.setName("Michele Li");
        System.out.println(a.getName());
        Student.printWelcomeMessage();
  }
}
```
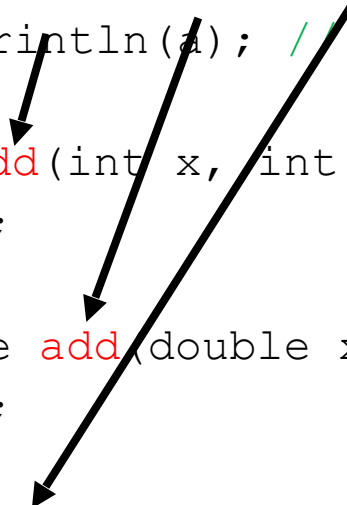
MyClass.java

```
public class Student{
  …
  public static void printWelcomeMessage(){
   …
  }
}
```

14

Overloaded Methods

Methods are said to be **overloaded** when there are multiple methods with the same name but a different signature.

```java
public class MyClass{
    public static void main(String[] args){
        double a = add(1, 2) + add(1.8, 5.2) + add(1, 2, 3);
        System.out.println(a); // 16.0
    }
    public static int add(int x, int y){
        return x + y;
    }
    public static double add(double x, double y){
        return x + y;
    }
    public static int add(int x, int y, int z){
        return x + y + z;
    }
}
```

Three overloaded methods named "add".

Printing objects

By default, Java doesn't know how to print objects:

```
Point p = new Point(10,7);
System.out.println("p is " + p);   // p is Point@9e8c34


// better, but cumbersome; p is (10, 7)
System.out.println("p is (" + p.getX() + ", " +
                              p.getY() + ")");



// desired behavior
System.out.println("p is " + p);   // p is (10, 7)
```

The `toString` method

The `toString` method tells Java how to convert an object into a String.

```
Point p1 = new Point(7, 2);
  System.out.println("p1: " + p1);

  // the above code is really calling the following:
  System.out.println("p1: " + p1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - Default: class's name @ object's memory address (base 16)

```
Point@9e8c34
```

toString **syntax**

`toString` can be overwritten to return a desired String representation of the object.

```
public String toString() {
        code that returns a String representing this object;
    }
```

- Method name, return, and parameters must match exactly.
- Example:

```
// Returns a String representing this Point.
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

Point class

```java
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
    // accessor methods
    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }
    public void translate(int dx, int dy) {
        setLocation(x + dx, y + dy);
    }
}
```

Client code

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

        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        System.out.println("p1: " + p1);
         //same as above
        System.out.println("p2: " + p2.toString());



}
}
```

OUTPUT:
```
p1:(5,2)
p2:(4,3)
```

Limitations of variables

- Idea: Make a variable to represent the size.
  - Use the variable's value in the methods.

- Problem: A variable in one method can't be seen in others.

```
public static void main(String[] args) {
    int size = 4;
    topHalf();
    printBottom();
}
public static void topHalf() {
    for (int i = 1; i <= size; i++) {      // ERROR: size not found
        ...
    }
}

public static void bottomHalf() {
    for (int i = size; i >= 1; i--) {      // ERROR: size not found
        ...
    }
}
```

Comments

Adding comments to your code helps to make it more readable and maintainable.

In the commercial world, software development is usually a team effort where many programmers will use your code and maintain it for years.

Commenting is essential in this kind of environment and a good habit to develop. Comments will also help you to remember what you were doing when you look back to your code a month or a year from now.

Comments

There are 3 types of comments in Java:

1) // Single line comment

2) /* Multiline comment */

3) /** Documentation comment */

We have seen the first two types of comments. The third is also a special version of the multi-line comment, /** */, called **the documentation comment.**

Java has a tool called javadoc that comes with the Java JDK that will pull out all of these comments to make documentation of a class as a web page.

Preconditions/Postconditions

A **precondition** is a condition that must be true for your method code to work, for example the assumption that the parameters have values and are not null. There is no expectation that the method will check to ensure preconditions are satisfied.

The methods could check for these preconditions, but they do not have to. The precondition is what the method expects in order to do its job properly.

A **postcondition** is a condition that is true after running the method. It is what the method promises to do. Postconditions describe the outcome of running the method, for example what is being returned or the changes to the instance variables.

Preconditions/Postconditions

/**

* Constructor that takes the x and y position of Sprite object

* Preconditions: parameters x and y are coordinates from 0 to

* the width and height of the window

*

* Postconditions: the Sprite object is placed in (x,y) coordinates

* @param x the x position to place the Sprite

* @param y the y position to place the Sprite */

```
public Sprite(int x, int y) {
        center_x = x;
        center_y = y;
}
```

Lab 1

Write a class called `MyComplex` which models the complex numbers a + bi. It contains:

1) Two private double variables `real` and `img`.

2) A default constructor to create a complex number at 0 + 0i.

3) A constructor which takes two double a and b and initializes this complex number to a + bi.

4) Setters and Getters(accessors and mutators) for private variables real and img.

5) `toString()` that returns "(a + bi") form of the complex number. If b is negative, we want the string to be "a - bi".

Lab 1

6) `isReal()` and `isImaginary()` that returns whether this complex number is real or imaginary, respectively. For example, 4 is real while 5i is imaginary but 4 + 5i is neither and 0 is both.

7) `void add(MyComplex c)`: Add complex number c to this complex number. Hint: (a+bi)+(c+di)=(a+c)+(b+d)i

8) `void multiply(MyComplex c)`: Multiply c to this complex number. Hint: (a+bi)*(c+di)=(ac-bd)+(ad+bc)i

9) `void conjugate()` changes this complex number into its conjugate. Hint: The conjugate of a + bi is a - bi.

10) `double argument()` returns the argument(angle) in radians of this complex number. This angle is the same as theta of polar coordinates. Hint: Use Math.atan2(y,x).

11) `double magnitude()` returns the magnitude or length of this complex number. Hint: The magnitude of a+bi is Math.sqrt(a*a+b*b). For example, the magnitude of 3+4i is 5.

Lab 1

MyComplex should also contains the following **static methods:**

1)`MyComplex addNew(MyComplex a, MyComplex b)` returns a complex number that is the sum of a and b.

2)`MyComplex multiplyNew(MyComplex a, MyComplex b)` returns the complex number that is the product a*b.

References

1) CPJava Website
2) CPJava Google Classroom
3) CPJava repl.it Classroom
4) Runestone CSAwesome BUSHSCHOOL_CPJAVA Course
5) Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp

References

1) CPJava Website
2) CPJava Google Classroom
3) CPJava trinket.io Classroom
4) Runestone CSAwesome BUSHSCHOOL_CPJAVA Course
5) Online Textbook Think Java - 2nd Edition by Allen Downey and Chris Mayfield
6) Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp