CS 113 – Computer Science I

Lecture 04 – Methods I

Thursday 09/14/2023
Announcements

• HW00 – due last night
• HW01 – releasing later today
  • Due Monday 09/18

• Read & Follow Instructions
  • Don’t just skim the labs & homework

• Office hours:
  • Monday: 2:45-4:00pm
  • Tuesday: 9:30-10:45am (I’ll try to get there by 9:15)
## Formatting Strings

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>Integer in base 10 (&quot;decimal&quot;)</td>
<td>12345</td>
</tr>
<tr>
<td>%,d</td>
<td>Integer with comma separators</td>
<td>12,345</td>
</tr>
<tr>
<td>%08d</td>
<td>Padded with zeros, at least 8 digits wide</td>
<td>00012345</td>
</tr>
<tr>
<td>%f</td>
<td>Floating-point number</td>
<td>6.789000</td>
</tr>
<tr>
<td>%.2f</td>
<td>Rounded to 2 decimal places</td>
<td>6.79</td>
</tr>
<tr>
<td>%s</td>
<td>String of characters</td>
<td>&quot;Hello&quot;</td>
</tr>
<tr>
<td>%x</td>
<td>Integer in base 16 (&quot;hexadecimal&quot;)</td>
<td>bc614e</td>
</tr>
</tbody>
</table>

Table 3.1: Example format specifiers
Demo

Demo 1: Ask user for a number, and return the square root

\[ \text{Math.sqrt(<number>)}; \]

Demo 2: Lets only print up to 2 decimal places

Demo 3: Lets round that answer to an integer
Math utilities

- `Math.round(40.11);`
- `Math.cos(0);`
- `Math.sqrt(9);`
- `Math.random();`

Examples of methods
Using methods

Abstraction:

allows us to use functionality without knowing how it works
Demo

Demo 1: Ask user for a number, and return the square root

```javascript
Math.sqrt(<number>);
```

Let's round that answer to an integer

Let's now do this for 2 numbers

Let's now do this for 4 numbers

Let's now do this for 6 numbers
Creating Methods

Idea: Define re-useable portions of code

Analogy: machines with inputs and outputs

Two steps for programming with functions:
1. Define the function (name, inputs, outputs, implementation)
2. Call the function with inputs and wait for its output

All methods should be contained inside a class
Anatomy of a method

• All methods have the following things:
  • Name
  • Parameter
  • Body
  • Return Type

```java
public static int method1 (int param1,
                       String param2) {
    /**
     * body of the method
     */
    return 0;
}
```
Method signature

public static int method1 (int param1, String param2)
Method documentation

/**
 * Description of the method
 * @param param1 description
 * @param param2 description
 * @return what the method returns
 */

public static int method1 (int param1,
                          String param2) {
    //...
Defining methods in Java: syntax

```java
public static void main(String[] args) {
    // function statements
}

public static float foo(int a, float b, String c) {
    // function statements
    System.out.println(c);
    return a*b;
}
```
Calling methods in Java: syntax

```java
public static float foo(int a, float b, String c) {
    // function statements
    System.out.println(c);
    return a*b;
}

public static void main(String[] args) {
    // function statements
    int value = 3;
    String c = "hello";
    float result = foo(value, -2.5, c);
    System.out.println(result);
}
```
Executing a method: steps

1. When you encounter a method, pause!
2. Create a *frame* to hold the method state
3. Copy argument values
4. Execute the method, line by line. Continue until
   1. you hit a return statement
   2. you run out of statements
5. Send back return value (can be nothing if function is *void*)
6. Delete the method’s frame
7. Resume original function
Exercise: Draw stack diagram

```java
public class Neg {

    public static double neg(double x) {
        double value = x * -1
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }

}
```
public class Neg {

    public static double neg(double x) {
        double value = x * -1
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }
}
Exercise: Draw stack diagram

```java
public class Neg {

    public static double neg(double x) {
        double value = x * -1
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }
}
```

Main

```
args: [ ]
absValue: 0
```
public class Neg {
    public static double neg(double x) {
        double value = x * -1;
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }
}

Exercise: Draw stack diagram
Exercise: Draw stack diagram

```java
public class Neg {

    public static double neg(double x) {
        double value = x * -1
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }
}
```
public class Neg {

    public static double neg(double x) {
        double value = x * -1
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }
}

Exercise: Draw stack diagram

Main

args:  
absValue: 0

abs

x:  
value  
return: 3.4

3.4
Exercise: Draw stack diagram

```java
public class Neg {
    public static double neg(double x) {
        double value = x * -1;
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }
}
```
Exercise: Draw stack diagram

```java
public class Neg {

    public static double neg(double x) {
        double value = x * -1
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }
}
```

![Stack diagram](image)
public class Neg {

    public static double neg(double x) {
        double value = x * -1;
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(-3.4);
    }
}
public class Neg {

    public static double neg(double x) {
        double value = x * -1
        return value;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = neg(5.4);
    }
}

Exercise: Draw stack diagram

Main
- args:
- absValue:

abs
- x:
- value
- return:
Exercise: Draw stack diagram

```java
public class Abs {

    public static double abs(double x) {
        if (x < 0) {
            return -x;
        }
        return x;
    }

    public static void main(String[] args) {
        double absValue = 0;
        absValue = abs(5.4);
    }
}
```

```
Main
  args: 
  absValue: 5.4

abs
  x: 5.4
    return value: 5.4
```
What is different here?

// Function: area
// Description: computes the area of a rectangle
// Input: width (double)
// Input: height (double)
// returns (double), the area as width * height
// side effects: none
public static double area(double width, double height) {
    return width * height;
}

public static double area(double width, double height) {
    return width * height;
}

// Function: area
// Description: computes the area of a rectangle
// Input: width (double)
// Input: height (double)
// returns (none)
// Side effect: prints the area to the console
public static void area(double width, double height) {
    double a = width * height;
    System.out.println("Area is "+ a);
}

public static void area(double width, double height) {
    double a = width * height;
    System.out.println("Area is "+ a);
}
Warning: don’t confuse printing with returning

// Function: area
// Description: computes the area of a rectangle
// Input: width (double)
// Input: height (double)
// returns (double), the area as width * height
// side effects: none
public static double area(double width, double height) {
    return width * height;
}

// Function: area
// Description: computes the area of a rectangle
// Input: width (double)
// Input: height (double)
// returns (none)
// Side effect: prints the area to the console
public static void area(double width, double height) {
    double a = width * height;
    System.out.println("Area is "+a);
}
Benefits of methods

• Split large problems into small problems

• Easier to maintain code/cleaner code
  • Only need to fix mistakes
  • DRY: Don’t repeat yourself

• Implement once, re-use in different programs

• Abstract details so user doesn’t need to worry about details
Method: IsInteger

$ java CheckInput
Enter an integer: apple
That is not an integer!!
Enter an integer: 0.0
That is not an integer!!
Enter an integer: 0-3
That is not an integer!!
Enter an integer: -4
You entered: -4

$ java CheckInput
Enter an integer:
That is not an integer!!
Enter an integer: 498756.0
That is not an integer!!
Enter an integer: 498756
You entered: 498756
Method specifications

**Idea:** “contract” between the function user and the method implementation

- Inputs and their types
- Return type
- Description of how function behaves, including special cases and side effects

A **side effect** refers to changes the method makes that last after the method returns (e.g. printing to the console is a side effect)

The **method signature** includes just the inputs and outputs of the function
Method Specifications

/**
* Returns a random real number from a Gaussian distribution with
* mean &mu and standard deviation &sigma
*
* @param mu the mean
* @param sigma the std
* @return a real number distributed according to the Gaussian distribution
* /
public static double gaussian(double mu, double sigma) {
    return mu + sigma * gaussian();
}
Why have method specifications?

• Make the behavior of function clear

• Enable user to use function without having to look at the implementation
Unit testing

Verify that method is implemented correctly

Call the method with different inputs and check the results

In a library, we can use the main method to test methods
Top down design

1. Identify features of the program
   1. List them out!

2. Identify verbs and nouns in feature list
   1. Verbs: functions
   2. Nouns: objects/variables

3. Sketch major steps – how features should fit together
   1. Algorithm!

4. Write program skeleton
   1. Include function stubs (placeholders for our functions)
   2. Function stub: empty function with parameters and return type

5. Implement and test function stubs one at a time