2.5 Methods

Calling or Invoking Methods

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like sin(π/2) and log(1/x). First, you evaluate the expression in parentheses, which is called the **argument** of the function. Then you can evaluate the function itself, maybe by punching it into a calculator.

The Java library includes a `Math` class that provides common mathematical operations. You can **call** or **invoke**, `Math` methods like this:

```
double root = Math.sqrt(17.0);
double angle = 1.5;
double height = Math.sin(angle);
```

The first line sets `root` to the square root of 17. The third line finds the sine of 1.5 (the value of `angle`).

Creating New Methods

We have explored how you can define new methods. By defining new methods, we can reuse code. Here's an example (System.out.println has been abbreviated to println):

```
public static void verse(String animal, String sound) {
  println("Old MacDonald had a farm, EE-I-EE-I-O,");
  println("And on that farm he had a " + animal + ", EE-I-EE-I-O,");
  println("With a " + sound + " " +sound + " here and a " + sound + " " +
          sound + "  there");
  println("Here a " + sound + ", there a " + sound + ", everywhere a " +
          sound + " " +sound);
  println("Old MacDonald had a farm, EE-I-EE-I-O.");
}

public static void singOldMacdonald() {
       verse("cow", "moo");
       System.out.println("");
       verse("dog", "woof");
}
```

Method names should begin with a lowercase letter and use "camel case", which is a cute name for jammingWordsTogetherLikeThis. You can use any name you want for methods, except main or any of the Java keywords.

In our example, verse and singOldMacdonald are public, but we won't worry about that right now. They are void, which means that they don't yield a result. They are merely a list of instructions to be carried out.

The parentheses after the method name can contain a list of variables, called **parameters**, where the method stores its arguments. Method verse has two parameters, animal and sound, which are both of type String. Method singOldMacdonald has no parameters.

You might wonder “But where is singOldMacdonald being called from?” So far, we have been using BlueJ to call methods manually, but later we will learn about the **main method** which is the start of any Java program.

<u>The Value of Putting Code Methods</u>

1. <u>Code Reuse:</u> An oft-used piece of code can be embedded in a method instead of typed out repeatedly. It can also be imported into other projects, saving a lot of time.

2. <u>Reduces errors:</u> If you are writing the same code repeatedly in many places in your program and suddenly decide to change it, you have it change it everywhere in your program and do it correctly each time. Each of these changes is a chance for making a mistake. If the calculation is embedded in a method, then you just change that one method, and the change happens the same everywhere.

3. <u>Code Readability:</u> By giving blocks of code a name, it simplifies your code and makes it easier to read.

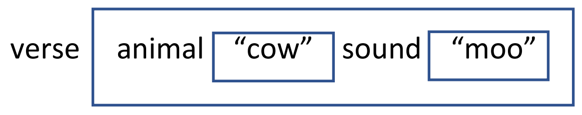
<u>Question 1:</u> What output is printed to the terminal when singOldMacdonald is invoked?

<u>Question 2:</u> How would you invoke the verse method if you wanted to sing about a duck that quacks?

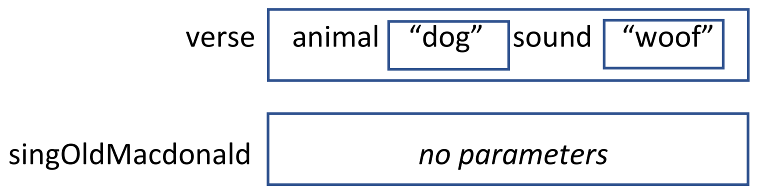
Stack Diagrams

As programs run, methods will invoke each other. An internal structure known as the Stack keeps track of what methods are being invoked. We can illustrate this with a Stack Diagram.

The **Stack Diagram** represents method invocations at a particular point in time, using boxes labelled with method names. Inside the box are more boxes that indicate the arguments that are passed to the parameters. So, the stack diagram for the invocation `verse(“cow”, “moo”)` would be:



A new method invocation may be put on the stack before another method finishes. This happens when a method invokes another method. As several invocations pile up, a stack diagram will include multiple invocations. The most recent invocations go at the top, while the oldest are at the bottom. For example, if we invoke `singOldMacdonald` and draw a stack diagram of the program at the moment after invoking `verse(“dog”, “woof”)`, we would see:



As methods finish, they are removed from the stack, and are no longer drawn in the Stack Diagram. In class we will take a look at an example of how the stack diagram changes as singOldMacdonald runs.

Question 3: What is the output of the following program? Be precise about where there are spaces and where there are newlines.

Hint: Start by describing in words what `ping` and `baffle` do when they are invoked.

```java
public static void zoop() {
    baffle();
    System.out.print("You wugga ");
    baffle();
}

public static void main(String[] args) {
    System.out.print("No, I ");
    zoop();
    System.out.print("I ");
    baffle();
}

public static void baffle() {
    System.out.print("wug");
    ping();
}

public static void ping() {
    System.out.println(".");
}
```

Question 4: Draw a stack diagram that shows the state of the program the first time `ping` is invoked.

Question 5: What happens if you invoke `baffle();` at the end of the `ping` method?

Question 6: Write the method displayBox that has two in parameters, width and height. The method will print out a box made of '$' characters to the terminal that is as wide and high as the arguments passed to it. For example, displayBox(6,3) would print this output to the terminal:

```
$$$$$$
$$$$$$
$$$$$$
```

Methods That Return Values

So far all of our methods in this assignment have had the “void” keyword in front of them. Void methods are just a group of statements that you want to run. However, like the CodingBat questions you tackled last week, you may have noticed that you can replace void with a type such as int, double, boolean, or String.

Methods that have a return type answer a question. An int method answers a numerical question like “what is the square root of 17?” and a boolean method answers a true or false question like “Is this party of squirrels successful, given these arguments?”

Here is an example of a method that calculates the distance between two points:

```
public static double dist (double x1, double y1, double x2, double y2)
{
   return Math.sqrt( (x2 – x1)*(x2 – x1) + (y2 – y1) * (y2 – y1));
}
```

Once Java encounters the return keyword, the method comes off the stack and the method call evaluates to whatever value was returned. So, if you had a statement like this:

```
     if ( dist(0, 0, 3, 4) < 10) {
          // ...
     }
```

When Java runs this if statement, it puts dist(0, 0, 3, 4) on the stack. Once it returns, dist evaluates to 5, which is less than 10, so the if statement condition is true. You could also do this:

```
     System.out.println(“The distance is “ + dist(0, 0, 3, 4));
```

dist will evaluate to 5, and then println will output:

```
      The distance is 5
```

Note that once you declare a method as having a return type, your method MUST return a value of that type or your program will not compile. For example, if you make a method like this:

```
public static int absoluteValue (int a) {
     if (a < 0) {
          return a*-1;
     }
}
```

This will not compile, as there is no return statement to handle values of a that are positive. You can fix it by having an else clause that handles all other values of a.

Question 7: Write a new dice rolling program that is similar to the Lesson 2.4 exercise 6, but change it so that instead of assuming they are rolling a 6-sided die, also ask how many sides the dice should have. In addition, put the dice rolling code into a method called "rollDie" that takes the number of sides of the dice as a parameter and returns a random integer in that range.

The output of such a program might look like this:

```
How many dice do you want to roll? 3
How many sides do these dice have? 10

You rolled: 2 5 9
Total: 16

Again? [y,n] y

How many dice do you want to roll? 2
How many sides do these dice have? 20

You rolled: 18 12
Total: 40

Again? [y,n] n
Good-bye!
```