What we will cover:
Call stacks
void methods
Pass by value

Call Stacks
As we have already seen, each time a method is invoked, memory is boxed off just for the execution of that method. The area of memory that holds all of this information is known as the stack. Stacks work in a last-in first-out (LIFO) fashion and have two main operations: push and pop. Items can be pushed onto the top of the stack and are popped off the top when finished. This enforces the LIFO order and gives the stack its name and structure. When a method is invoked, memory is allocated for the variables and data in that method and that memory space is pushed onto the stack. When the code is finished executing, the memory space is popped from the stack and released. The most recently called code is executed first and once it is done executing, the control of the program is returned to the caller and any remaining code is executed.

Each successive call to a method will create another space on top of the stack. Other methods (user-created or from an existing library) can be called from within methods as we have seen with Math.sqrt(...); new space will simply be created for the method’s variables and data, that spaced will be pushed onto the stack, and popped off the stack when finished executing. As has been noted, you can view this occurring when using a debugger during the execution of the method.

Class Exercise 20 – Call stacks and method execution

void methods
Remember that procedures or void methods return no value, but perform some kind of task or action. These methods are invoked using standard statements.

You can have a return statement in a void method to immediately end the execution of the method. That will pop the method’s space off the stack, returning control to the caller. This is not really recommended, but still possible. You simply issue the command return followed by a semicolon (return; ) to exit immediately out of the void method. Notice that proper conditional statements can prevent having to use this, but it is available if necessary. This works similar to the break statement that is used in switches and loops.
Review the following code that prints out the letter grade instead of returning it for all the values (by 5s) between -10 and 110.

```java
public class PrintGrades {
    public static void printGrade(double grade) {
        if (grade >= 0 && grade <= 100) {
            System.out.print(grade + " is a(n) ");
            if (grade >= 90) {
                System.out.println("A");
            } else if (grade >= 80) {
                System.out.println("B");
            } else if (grade >= 70) {
                System.out.println("C");
            } else if (grade >= 60) {
                System.out.println("D");
            } else {
                System.out.println("F");
            }
        } else {
            System.out.println("That is not a valid grade");
        }
    }

    public static void main(String[] args) {
        int start = -10;
        int end = 110;
        int step = 5;
        for (int i = start; i <= end; i+=step) {
            printGrade(i);
        }
    }
}
```

We could create a method that works over a range of numbers and prints the results of another method call:

```java
public class FactorialRange {
    public static void factorialRange(int start, int end) {
        for (int i = start; i <= end; i++) {
            System.out.println(i + "! = " + factorial(i));
        }
    }

    public static int factorial(int number) {
        int product = 1;
        for (int i = 1; i <= number; i++) {
            product *= i;
        }
        return product;
    }

    public static void main(String[] args) {
        factorialRange(0, 7);
    }
}
```
We could convert our `isEven()` method into a void method that prints a result directly from the method:

```java
public static void isEven(double number) {
    String output = "The number " + number + " is ";

    if (number % 2 == 0) {
        output += "even."
    } else {
        output += "odd."
    }

    System.out.println(output);
}
```

**Pass by value**

Parameters in Java are passed by value as opposed to passed by reference. When methods are invoked, only the values of the parameters, whether or not they are variables, are passed to the method’s memory space. The original values of these variables are not changed. Those values get assigned to new variables (the parameters) inside the method’s stack space. A method call does not change a main method’s value unless the value of the returning result is placed in the main method’s variable.

The following Java code does not actually swap the variables inside the main method. Inside of the swap method, the values are swapped, but those variables go away when the method is done executing.

```java
public class PassByValue {

    public static void swap(int i, int j) {
        System.out.println("\tinside swap, before swap, i = " + i + " j = " + j);
        int temp = i;
        i = j;
        j = temp;
        System.out.println("\tinside swap, after swap, i = " + i + " j = " + j);
    }

    public static void main(String[] args) {
        int i = 2, j = 5;
        System.out.println("before swap, i = " + i + " j = " + j);
        swap(i, j);
        System.out.println("after swap, i = " + i + " j = " + j);
    }
}
```

This code produces the following output:

```
before swap, i = 2 j = 5
    inside swap, before swap, i = 2 j = 5
    inside swap after swap, i = 5 j = 2
after swap, i = 2 j = 5
```

Notice that inside the `swap()` method, the numbers have been swapped, but remember the call stack, all of that memory goes away once the method is done executing, so any values that were contained in that memory are now gone.
A good way to demonstrate this is to look at code in a language that does pass allow pass by reference, C++. The following code shows the difference between pass by value and pass by reference.

```cpp
#include <iostream>

using namespace std;

void swap(int i, int j) {
    int temp = i;
    i = j;
    j = temp;
}

void swapRef(int& i, int& j) {
    int temp = i;
    i = j;
    j = temp;
}

int main(int argc, char **argv) {
    int i = 2, j = 5;
    cout << "i is " << i << " j is " << j << endl << endl;
    swap(i, j);
    cout << "i is " << i << " j is " << j << endl << endl;
    swapRef(i, j);
    cout << "i is " << i << " j is " << j << endl << endl;
    system("pause");
    return 0;
}
```

This code produces the following output:

```
i is 2 j is 5
i is 2 j is 5
i is 5 j is 2
Press any key to continue . . .
```
Let's look at the code with detailed output statements:

```cpp
#include <iostream>

using namespace std;

void swap(int i, int j) {
    cout << "\t\tin swap, before swap, i is " << i << " j is " << j << endl;
    int temp = i;
    i = j;
    j = temp;
    cout << "\t\tin swap, after swap, i is " << i << " j is " << j << endl;
}

void swapRef(int& i, int& j) {
    cout << "\t\tin swapRef, before swap, i is " << i << " j is " << j << endl;
    int temp = i;
    i = j;
    j = temp;
    cout << "\t\tin swapRef, after swap, i is " << i << " j is " << j << endl;
}

int main(int argc, char **argv) {
    int i = 2, j = 5;

    cout << "pass by value" << endl << "---------" << endl;
    cout << "in main before swap i is " << i << " j is " << j << endl;
    swap(i, j);
    cout << "in main after swap i is " << i << " j is " << j << endl << endl;

    cout << "pass by reference" << endl << "---------" << endl;
    cout << "in main before swap i is " << i << " j is " << j << endl;
    swapRef(i, j);
    cout << "in main after swap i is " << i << " j is " << j << endl << endl;

    system("pause");
    return 0;
}
```

This code produces the following output:

```
pass by value
In main before swap i is 2 j is 5
In swap, before swap, i is 2 j is 5
In swap, after swap, i is 5 j is 2
In main after swap i is 5 j is 5

pass by reference
In main before swap i is 2 j is 5
In swapRef, before swap, i is 2 j is 5
In swapRef, after swap, i is 5 j is 2
In main after swap i is 5 j is 2

Press any key to continue . . .
```

The normal `swap()` function takes parameters as values and does not change them in the main method. The `swapRef()` function however, simply passes references to the original variables. Any changes that happen in the method actually change the values, in memory, of the original values.