Lecture 9
Call-by-Value* and Recursion

*When is a Method not like a Function?
Call-by-Value

Can a method change the data in its argument list?

```java
public class CallByValue {
    public static void main(String[] args) {
        int n;
        n = 37;
        System.out.println("before: n = " + n);
        SetToSix(n);
        System.out.println("after: n = " + n);
    }

    static void SetToSix(int n) {
        System.out.println("inside: n = " + n);
        n = 6;
    }
}
```

The value of n in `SetToSix` is initially set to the value of the corresponding variable on the argument list in the method from which the call originated. In this case 37.

Can you guess the last line of output?

- before: n = 37
- inside: n = 37
- after: n = __
public class CallByValue
{
    public static void main(String[] args)
    {
        int n;
        n = 37;

        System.out.println("before: n = " + n);
        SetToSix(n);
        System.out.println("after: n = " + n);
    }

    static void SetToSix(int n)
    {
        System.out.println("inside: n = " + n);
        n = 6;
    }
}
What about Global Variables?

Can you guess the output?

```java
public class CallByValue {
    static int N; // a GLOBAL variable

    public static void main(String[] args) {
        N = 314;
        System.out.println("N = " + N);
        SetToSix();
        System.out.println("N = " + N);
    }

    static void SetToSix() {
        N = 6;
    }
}
```

The argument list is empty. Would the answer be the same if N had been passed as an argument?
public class CallByValue
{
    static int N;

    public static void main(String[] args)
    {
        N = 314;
        System.out.println("N = " + N);
        SetToSix();
        System.out.println("N = " + N);
    }

    static void SetToSix()
    {
        N = 6;
    }
}

CallByValue

N

0
314
6
6
What about Arrays?

```java
public class CallByValue {
    public static void main(String[] args) {
        double[] x = new double[12];
        for (int j=0; j<12; j++) { x[j] = j; }
        System.out.print("before: x[7] = " + x[7]);
        ComputeSquares(x);
        System.out.println("after: x[7] = " + x[7]);
        System.out.print("before: x[4] = " + x[4]);
        ComputeSqrt(x[4]);
    }
    static void ComputeSquares(double[] ex) {
        for (int j=0; j<12; j++) {
            ex[j] = ex[j]*ex[j];
        }
    }
    static void ComputeSqrt(double y) {
        y = Math.sqrt(y);
    }
}
```

What do you think is the output this time?


**Moral:** The *value* that is copied to the called method is a pointer, not the entire pile of stuff to which the pointer points.
Arrays: Memory View

public class CallByValue
{
    public static void main(String[] args)
    {
        double[] x = new double[4];
        for (int j=0; j<4; j++) {
            x[j] = j;
        }
        ComputeSquares(x);
        ComputeSqrt(x[2]);
    }
    static void ComputeSquares(double[] ex)
    {
        for (int j=0; j<12; j++) {
            ex[j] = ex[j]*ex[j];
        }
    }
    static void ComputeSqrt(double y)
    {
        y = Math.sqrt(y);
    }
}
What about Class Variables?

class Zip
{
    int zip; double lat; double lon; double x; double y;
}
public class CallByValue
{
    public static void main(String[] args)
    {
        Zip z;
        z = new Zip();
        z.zip = 90210;
        z.lat = 34.09;
        z.lon = 118.41;
        System.out.println("before: zip = " + z.zip);
        SetToPrinceton(z);
        System.out.println("after: zip = " + z.zip);
    }
    static void SetToPrinceton(Zip zed)
    {
        zed.zip = 8544;
    }
}

Moral: Class variables are pointers just as arrays are. Therefore, argument passing works the same as it did with arrays.
class Zip
{
    int zip; double lat; double lon;
}

class CallByValue
{
    public static void main(String[] args)
    {
        Zip z;

        z = new Zip();
        z.zip = 90210;
        z.lat = 34.1;
        z.lon = 118.;
        SetToPrinceton(z);
    }

    static void SetToPrinceton(Zip zed)
    {
        zed.zip = 8544;
    }
}
Recursion

Consider the \textit{factorial} function: \( n! = n(n-1) \cdots (3)(2)(1) \)

The \textit{natural} way to program it uses the following recursive definition:

Here’s the Java program:

\[
\begin{align*}
\text{import ccj.*;} \\
\text{public class Factorial} \\
\text{    \{ } \\
\text{        public static void main(String[] args)} \\
\text{        \{ } \\
\text{            int n, n_fact; } \\
\text{            while (true) } \\
\text{                n = Console.in.readInt("Enter an integer: "); } \\
\text{                if (n<0) break; } \\
\text{                n_fact = factorial(n); } \\
\text{                System.out.println("n factorial = "+n_fact); } \\
\text{            } \\
\text{        } \\
\text{    \}} \\
\text{    static int factorial(int k)} \\
\text{        \{ } \\
\text{            if (k==0) \{return 1;\}} \\
\text{            else \{return k*factorial(k-1);\}} \\
\text{        } \\
\text{    } \\
\end{align*}
\]
Digression -- Factorials and Integration

Amazing but true: \[ n! = \int_0^\infty x^n e^{-x} \, dx \]

Can use a slightly modified `Integra.java` to compute these integrals:

Works even with positive real numbers and even some negative numbers:

\[ (-1/2)! = \sqrt{\pi} \]