Lecture 9
Call-by-Value* and Recursion

*When is a Method not like a Function?

“Just a darn minute! Yesterday you said \( x \) equals two!”
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?

```plaintext
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;
    }
}

Main
n
SetToSix
n
<table>
<thead>
<tr>
<th>Main</th>
<th>SetToSix</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>6</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>37</td>
<td>6</td>
</tr>
</tbody>
</table>
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;
    }
}
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

```java
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);
    }
}
```

```java
main  
0     344   0     0     0
0     344   344   352   360   368
344   0     1     2     3
344   0     1     4     9
344   0     1     4     9
```

```java
ComputeSquares  
344  556   0     1     4     9
```

```java
ComputeSqrt  
y
4
712
2
```
What about Objects?

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;
    }
}

And the output is

before: zip = 90210
after:  zip = _____

Moral: Class variables are pointers just as arrays are. Therefore, argument passing works the same as it did with arrays.
Objects: Memory View

class Zip
{
  int zip; double lat; double lon;
}
public 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;
  }
}
Static Variables

```java
class SomeStuff {
    static int s;
    double value;
}
```

Static variable `s` belongs to the class `SomeStuff`. There is always exactly one "copy of this variable."

Variable `value` belongs to instances of class `SomeStuff`.
- Zero or more instances of this variable can exist at any time.
- They are created by calls to `new SomeStuff()`.

Terminology:
- Static variables are called `class variables`.
- Other variables are called `instance variables`.
A Demo Program

```java
public class StaticDemo {
    public static void main(String[] args) {
        SomeStuff x, y;
        x = new SomeStuff();
        y = new SomeStuff();

        x.s = 1;
        x.value = 3.14159;

        y.s = 2;
        y.value = 2.71828;

        System.out.println("x.s     = " + x.s);
        System.out.println("x.value = " + x.value);

        System.out.println("y.s     = " + y.s);
        System.out.println("y.value = " + y.value);

        SomeStuff.s = 3;
        System.out.println("x.s     = " + x.s);
    }
}
```

What’s the output?

Referring to class variable `s` using `x.s` is legal but misleading.

`x.s` and `y.s` are the same variable.

It is better to use `SomeStuff.s`.

Writing `SomeStuff.value` is illegal - i.e. it produces a compile-time error.

<table>
<thead>
<tr>
<th>x.s</th>
<th>___</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.value</td>
<td>___</td>
</tr>
<tr>
<td>y.s</td>
<td>___</td>
</tr>
<tr>
<td>y.value</td>
<td>___</td>
</tr>
<tr>
<td>x.s</td>
<td>___</td>
</tr>
</tbody>
</table>
Recursion

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

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

Here’s the Java program:

\[
\begin{align*}
\text{import cs1.*;} \\
\text{public class Factorial} \\
\text{  \{} \\
\text{    public static void main(String[] args)} \\
\text{    \{} \\
\text{      int n, n_fact; \\
\text{      while (true)} \\
\text{        \{} \\
\text{          n = Keyboard.readInt("Enter an integer: "); \\
\text{          if (n<0) break; \\
\text{          n_fact = factorial(n); \\
\text{          System.out.println("n factorial = "+n_fact);} \\
\text{          \} \\
\text{      \} \\
\text{      System.out.println("n factorial = "+n_fact);} \\
\text{  \} \\
\text{  } \\
\text{static int factorial(int k)} \\
\text{  \{} \\
\text{    if (k==0) \{return 1;} \\
\text{    else \{return k*factorial(k-1);\}} \\
\text{  \} \\
\text{\}}
\end{align*}
\]
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:

![Integration of: a x^k exp(-x/b)](image)

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

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