Static Methods

Anatomy, figure page 188.
Class.name or just “name”
Scope, page 189.
Overloading

```java
public static int abs (final int x) {
    if (x<0) return -x else return x;
}
public static double abs (final double x) {
    if (x<0.0) return -x else return x;
}
public static int abs (final int x) {
    return x<0?-x:x;
}
```
A class can contain (static) code which can be used by another class. For example, one might use `Math.hypot`.
See page 219, in S&W.
Parameter Passing

The names of the parameters to a subprocedure should be used to refer only the arguments, and, so, should be declared to be final in Java. This is hardly ever any exception to the rule. Occasionally one needs an exception in the case of local variables, but hardly ever for the case of formal parameters.
public class Main
{
    public static int multiply (final int x, final int y)
    {
        final int p = 5*(x-1);
        return p*y;
    }

    public static void main (final String[] args) {
        final int a = Integer.parseInt (args[0]);
        final int ans = multiply (a, a+3);
    }
}

public class Main {
    public static void main (final String[] args) {
        final int a = Integer.parseInt (args[0]);
        multiply: {
            final int x = a;   -- assign actual 1 to formal
            final int y = a+3; -- assign actual 2 to formal
            final int p = 5*(x-1); -- body of procedure
        }
        final int ans = p*y;
    }
}
To understand a procedure call conceptually, the body of the subprocedure is inserted in-line with the code of the caller. Of course, it is not practical or desirable to eliminate all subprocedures in the source code. And, in the case of recursion, it is not possible to eliminate the calls since the process would not terminate. Fortunately, recursion is much easier to understand.
Recursion
Recursion

Anything that, in happening, causes itself to happen again, happens again.

Douglas Adams, *Mostly Harmless*
Recursion

Sedgwick & Wayne, *Introduction to Programming in Java*, Section 2.4.
Adams, Nyhoff, Nyhoff, *Java*, Section 8.6, page 457.
Skansholm, *Java From the Beginning*, Section 15.4, page 488.
Main, *Data Structures Using Java*, Chapter 8, page 371.
Problems can be decomposed into smaller problems and their solutions composed into the solution of the original problems. In many cases, the smaller problem is the same as the original problem. As long as this problem can be easily solved for some inputs, a recursive solution has been found.

A recursive solution has the advantage of reusing the same technique (with different size inputs) and so fewer subprocedures need be written.

All problems can be solved this way; difficult problems can have elegant solutions when solved this way.
Value of Recursion

Powerful and clear mechanism to solve difficult problems.

Unfortunately, few are careful at employing such a powerful mechanism and mathematics does not distinguish between computationally smart recursion and computationally inefficient recursion.
A **recursive call** is a subprogram call to the program making the call.

Anatomy of recursion

- **Base case.** A simple case solved immediately without requiring recursion.
- **General case.** A case for many inputs solved with recursion.
- **Smaller-caller issue.** Each recursive call must involve a “smaller” problem or one “closer” to the base case.
Recursive Definitions of Functions

Mathematics is lousy with recursive definitions.

These definitions are very easy to translate into Java. (But mathematicians don’t care about computational efficiency.)
Recursive Definitions of Functions

\[ n! = \begin{cases} 
1 & \text{if } n = 0, \\
 n \cdot (n-1)! & \text{otherwise} 
\end{cases} \]

\[ x^n = \begin{cases} 
1 & \text{if } n = 0, \\
x \cdot x^{(n-1)} & \text{otherwise} 
\end{cases} \]

\[ \text{fib}(n) = \begin{cases} 
1 & \text{if } n = 1, \\
1 & \text{if } n = 2, \\
\text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} 
\end{cases} \]

For \( x \geq y \),

\[ \text{gcd}(x, y) = \begin{cases} 
x & \text{if } y = 1, \\
\text{gcd}(y, x \mod y) & \text{otherwise} 
\end{cases} \]
Recursive Definitions of Functions

For $0 \leq r \leq n$, 
\[
C(n, r) = \begin{cases} 
1 & \text{if } n = 1 \text{ or } n = r, \\
C(n-1, r-1) + C(n-1, r) & \text{otherwise}
\end{cases}
\]

For $0 \leq k \leq n$, 
\[
s(n, k) = \begin{cases} 
1 & \text{if } n = k, \\
0 & \text{if } k = 0, \\
s(n-1, k-1) + (n-1) \cdot s(n-1, k) & \text{otherwise}
\end{cases}
\]

For $0 \leq k \leq n$, 
\[
S(n, k) = \begin{cases} 
1 & \text{if } n = k, \\
0 & \text{if } k = 0, \\
S(n-1, k-1) + k \cdot S(n-1, k) & \text{otherwise}
\end{cases}
\]
Recursive Definitions of Functions

\[ Ack(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0, \\
  Ack(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0, \\
  Ack(m - 1, Ack(m, n - 1)) & \text{otherwise}
\]
Factorial

\[ n! = \begin{cases} 
1 & \text{if } n = 0, \\
 n \cdot (n-1)! & \text{otherwise} 
\end{cases} \]

fact(0) = 1
fact(n) = n * fact(n-1)

```java
import java.util.Arrays;

public class Main {

    public static int fact(final int n) {
        if (n == 0) return 1;
        else return n * fact(n - 1);
    }

    public static void main(final String[] args) {
        System.out.println(fact(4));
        System.out.println(fact(7));
    }

}
```
Factorial

```java
int fact (final int n) {
    if (n==0) return 1;
    else return n * fact(n-1);
}
```

One might accept the clumsy:

```java
int fact (final int n) {
    final int ret;
    if (n==0) ret=0;
    else ret = n * fact(n-1);
    return ret;
}
```

But never the error prone:

```java
int fact (final int n) {
    int ret=1;  // Can’t be final!
    if (n>0) ret = n * fact(n-1);
    return ret;
}
```
Factorial

To correctly capture the idea of a recursive solution:

\[
  n! = \begin{cases} 
    1 & \text{if } n = 0, \\
    n \cdot (n-1)! & \text{otherwise}
  \end{cases}
\]

we realize the following identities hold:

\[
  \text{fact}(n) = 1 \quad -- \quad n=0 \\
  \text{fact}(n) = n \cdot \text{fact}(n-1) \quad -- \quad n>0
\]

which is clearly realized in the Java code:

```java
int fact (final int n) {
    if (n==0) return 1;
    else return n * fact(n-1);
}
```
Exponential

\[ x^n = \begin{cases} 
1 & \text{if } n = 0, \\
x \cdot x^{(n-1)} & \text{otherwise} 
\end{cases} \]

```java
int exp (final int x, final int n) {
    if (n==0) return 1;
    else return x * exp (x, n-1);
}
```
Fibonacci

$$fib(n) = \begin{cases} 
1 & \text{if } n = 1, \\
1 & \text{if } n = 2, \\
fib(n-1) + fib(n-2) & \text{otherwise}
\end{cases}$$

```java
int fib (final int n) {
    if (n==0) return 0;
    else if (n==1) return 1;
    else return fib(n-1) + fib(n-2)
}
```
int gcd (final int p, final int q) {
    if (q==0) return p;
    else return gcd (q, p%q);
}
Binomial coefficient

For $0 \leq r \leq n$,

$$C(n, r) = \begin{cases} 1 & \text{if } r = 0 \text{ or } r = n, \\ C(n-1, r-1) + C(n-1, r) & \text{otherwise} \end{cases}$$

$c(n, 0) = 1$
$c(n, r) = \text{if } (n==r) \text{ then } 1 \text{ else } c(n-1, r-1) + c(n-1, r)$

```java
int binomial (final int n, final int r) { 
    if (r==0 || r==n) return 1;
    else return binomial (n-1,r-1) + binomial (n-1,r);
}
```
Stirling Numbers of the First Kind

For $0 \leq k \leq n$,

$$s(n, k) = \begin{cases} 
1 & \text{if } n = k, \\
0 & \text{if } k = 0, \\
s(n-1, k-1) + (n-1) \cdot s(n-1, k) & \text{otherwise}
\end{cases}$$

```java
int stirling1 (final int n, final int k) {
    if (n==k) return 1;
    else if (k==0) return 0;
    else return stirling1 (n-1,k-1) + (n-1)*stirling1 (n-1,k);
}
```
Stirling Numbers of the Second Kind

For \(0 \leq k \leq n\),

\[
S(n,k) = \begin{cases} 
1 & \text{if } n = k, \\
0 & \text{if } k = 0, \\
S(n-1,k-1) + k \cdot S(n-1,k) & \text{otherwise}
\end{cases}
\]

```java
int stirling2 (final int n, final int k) {
    if (n==k) return 1;
    else if (k==0) return 0;
    else return stirling2 (n-1,k -1) + k*stirling2 (n-1,k);
}
```
Ackermann Function

A well-known, fast-growing function.

\[ Ack(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0, \\
  Ack(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0, \\
  Ack(m - 1, Ack(m, n - 1)) & \text{otherwise}
\end{cases} \]

```java
int ackermann (final int m, final int n) {
    if (m==0) return n+1;
    else if (m>0 && n==0) return ackerman (m-1,1)
    else return ackermann (m-1, ackerman (m,n-1));
}
```
More Recursive Functions

```java
boolean isPalindrome (final String s) {
    final int len = s.length ();
    if (len < 2) {
        return true;
    } else {
        return s.charAt (0) == s.charAt (len-1) &&
               isPalindrome (s.substring (1, len-1));
    }
}
```
public static boolean lexicographic(String x, String y) {
    if (y.length() == 0) return false;
    else if (x.length() == 0) return true;
    else if (x.charAt(0) < y.charAt(0)) return true;
    else if (x.charAt(0) > y.charAt(0)) return false;
    else return lexicographic(x.substring(1), y.substring(1));
}
More Recursive Functions

int largest (int[] a) {
    return largest (a, 0, Integer.MIN_VALUE);
}

int largest (int[] a, int start, int max) {
    if (start == a.length) {
        return max;
    } else {
        final int l = a[start] > max ? a[start] : max;
        return largest(a, start + 1, l);
    }
}
double sin (final double x) {
  if (Math.abs (x) < 0.005) {
    return x - x*x*x/6.0;  // An approximation for
  } else {
    return 2.0 * sin(x/2.0) * cos(x/2.0);
  }
}

double cos (final double x) {
  if (Math.abs (x) < 0.005) {
    return 1.0 - x*x/2.0;  // An approximation for
  } else {
    return 1.0 - 2.0*sin(x/2.0)*sin(x/2.0);
  }
}
public class Main

    public static int fact (final int n) {
        if (n==0) return 1;
        else return n * fact(n-1);
    }

    public static void main (final String[] args) {
        System.out.println (fact(4));
    }
}
\[
\begin{align*}
\text{fact (0)} &= 1 \\
\text{fact (n)} &= n \times \text{fact (n-1)} \\
\text{fact 4} &= \\
&= 4 \times \text{fact (3)} \\
&= 4 \times 3 \times \text{fact (2)} \\
&= 4 \times 3 \times 2 \times \text{fact (1)} \\
&= 4 \times 3 \times 2 \times 1 \times \text{fact (0)} \\
&= 4 \times 3 \times 2 \times 1 \times 1 \\
&= 4 \times 3 \times 2 \times 1 \\
&= 4 \times 3 \times 2 \\
&= 4 \times 6 \\
&= 24
\end{align*}
\]

The “left-over” work requires a lot of memory which is unfortunate (and unnecessary).
// Compute n!*r
public static int fact (final int n, final int r) {
    if (n==0) return r;
    else return fact(n-1, n*r);
}

This has the tremendous advantage of being tail recursive (the recursive call is the last thing the function does). Such a recursive function can be translated in a loop by the compiler avoiding the overhead of a procedure call to store the left-over work.

fact (4,1) =
    = fact (3,4)
    = fact (2,12)
    = fact (1,24)
    = fact (0,24)
    = 24
public class Main
    
    public static int gcd (final int p, final int q)
        if (q==0) return p;
        else return gcd (q, p%q);
    }
    
    public static void main (final String[] args) {
        System.out.println (gcd (1272, 216));
    }
}

gcd (p, 0) = p
gcd (p, q) = gcd (q, p%q);

gcd (1272, 216) =
    = gcd (216, 192)
    = gcd (192, 24)
    = gcd (24, 0)
    = 24
Recursive Definitions of Functions

The number of committees of \( r \) members from a total of \( n \) people:

\[
C(n, r) = \begin{cases} 
1 & \text{if } r = 0 \text{ or } r = n, \\
C(n - 1, r - 1) + C(n - 1, r) & \text{otherwise}
\end{cases}
\]

More generally we allow \( r = 0 \) and \( n = 0 \) and define the number of combinations as the number of ways a subset of \( r \leq n \) items can be chosen out of a set of \( n \geq 0 \) items. The function is sometimes known as the choose function and the value as the binomial coefficient.

\[
C(n, r) = C_r^n = nC_r = \binom{n}{r} = \frac{n!}{r!(n-r)!}
\]
Arranging the binomial coefficients in a triangle, gives us Pascal’s famous triangle:

<table>
<thead>
<tr>
<th>n/r</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>15</td>
<td>20</td>
<td>15</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>
Binomial Coefficient

The formula

\[ C(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!} \]

is interesting, but very bad for computation as the factorial grows very large even if the answer is small.

The Java program has lots of recomputation:

```java
// Compute: n choose r
int choose (final int n, final int r) {
    if (n==0 || n==r) return 1;
    else if (r==1) return n;
    else return choose(n-1, r -1) + choose(n-1, r);
}
```
Binomial Coefficient

Mathematical identities about the binomial coefficient.

\[
\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}
\]

\[
\binom{n}{k} = \binom{n}{n-k}
\]

\[
\binom{n}{k} = \frac{n!}{(n-k)!k!}
\]

\[
\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}
\]
Sometimes the easiest way to conceive it is not the best way to program it. Here is an equivalent version that is more efficient.

\[
C(n, r) = \begin{cases} 
1 & \text{if } n = 1 \text{ or } n = r, \\
n & \text{if } r = 1, \\
\frac{n}{r} C(n - 1, r - 1) & \text{otherwise}
\end{cases}
\]

// Compute: \( n \) choose \( r \)
public static int choose (final int n, final int r) {
    if (n==0 || n==r) return 1;
    else if (r==1) return n;
    else return n * choose(n-1, r -1) / r;
}
Sometimes the solving a more general problem presents a solution which is more efficient.

\[ C(n, r, a) = \begin{cases} 
  a & \text{if } r = 0 \text{ or } n = r, \\
  C(n-1, r-1, a \times n/k) & \text{otherwise}
\end{cases} \]

// Compute: \( a \times (n \text{ choose } r) \)
```java
public static int choose (final int n, final int r, final int a) {
    if (r==0 || n==r) return 1;
    else return choose(n-1, r-1, a*n/r);
}
```
Recursive Definitions of Functions

The number of ways to form \( k \) (non-empty) teams from a total of \( n \) people (everybody is on one of the teams):

\[
s(n, k) = \begin{cases} 
1 & \text{if } n = 1 \text{ or } n = k, \\
1 & \text{if } k = 1, \\
s(n - 1, k - 1) + k \cdot s(n - 1, k) & \text{otherwise}
\end{cases}
\]

To understand the recurrence, observe that a person “\( n \)” is on a team by himself or he is not. The number of ways that “\( n \)” is a team by himself is \( s(n - 1, k - 1) \) since we must partition the remaining \( n - 1 \) people in the the available \( k - 1 \) teams. The number of ways that “\( n \)” is a member of one of the \( k \) teams containing other people is given by \( k \cdot s(n - 1, k) \).
Stirling numbers of the second kind $\left\{{n \atop k}\right\}$

<table>
<thead>
<tr>
<th>$n/k$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>15</td>
<td>25</td>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>31</td>
<td>90</td>
<td>65</td>
<td>15</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>63</td>
<td>301</td>
<td>350</td>
<td>140</td>
<td>21</td>
<td>1</td>
</tr>
</tbody>
</table>

Stirling numbers of the second kind (sequence A008277 in OEIS):

$$1, 1, 1, 1, 3, 1, 1, 7, 6, 1, 1, 15, 25, 10, 1, \ldots$$
Recursive Definitions of Functions

The number of expressions containing $n$ pairs of parentheses which are correctly matched. For $n = 3$ we have five: $$(((())), ()(()), ()(()) , (()()), (()()))$$

$$C_n = \begin{cases} 
1 & \text{if } n = 0 \\
\frac{2(2n-1)}{n+1} C_{n-1} & \text{otherwise}
\end{cases}$$

The first Catalan numbers (sequence A000108 in OEIS) are

$$1, 1, 2, 5, 14, 42, 132, \ldots$$
Recursion is always easy to understand

Why? No time is involved.
Recursion is easy to understand and easy to do. It possible to very powerful things with recursion. In this way it is surprisingly easy to exceed the ability of the computer.

- stack overflow
- excessive recomputation
You can do anything with recursion.

```c
int add (int n, int m) {
    if (m==0) return n;
    else add (n, m-1);
}
```
```c
int fib (int n) {
    if (n==0) return 0;
    else if (n==1) return 1;
    else return fib(n-1)+fib(n-2)
}

Makes a simple and easy definition, but hidden in the computation is
the wasteful recomputation of my common values.

fib2 0 = (1,0)
fib2 1 = (1,1)
fib2 n = let (a,b)=fib2 n-1 in (a+b, a)

int fib3 (int n, int a, int b) {
    if (n==0) return a;
    else fib3 (n-1, a+b, a)
}
```
It is easier to make a correct program more efficient than to make a buggy program more correct.

A program that does what you think it does is much better than a program that might do what you want it do.