

# Example of a Java program

```
class SomeNumbers
{
    static int square (int x)
    {
        return x*x;
    }

    public static void main (String[] args)
    {
        int n=20;
        if (args.length > 0) // change default
            n = Integer.parseInt(args[0]);
        for (int i=0; i <= n; i++)
        {
            System.out.print("The square of " + i + " is ");
            System.out.println(square(i));
        }
    }
}
```

# Compiling and running the Java program

```
$ javac SomeNumbers.java
$ ls Some*
SomeNumbers.class  SomeNumbers.java
$ java SomeNumbers 10
The square of 0 is 0
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
$
```

# Class definition

```
[access_modifier] class class_name
{
    method definition 1
    method definition 2
    method definition 3
    method definition 4

    // etc.
}
```

## **access\_modifier:**

(optional except for main method, which is **public**): one of

- public** - accessible to every user of the class
- private** - accessible only inside class
- protected** - accessible only to classes in same directory (default if nothing written)

**class\_name:** **identifier**, i.e. lower or Upper case letter followed by letters, digits, or underscore "\_". Class names start in Upper case by convention.

# Method definition

```
[access_modifier] [static] return_type method_name (parameter_list)
{
    statement 1 ;
    statement 2 ;
    statement 3 ;
    statement 4 ;

    // etc.

    return expression ;    // except if return type is void
}
```

**return\_type** (mandatory): any data type, e.g. one of

**int** **boolean** **double** **float** **long** **char**

- primitive types

**void** - nothing to be returned (method is not a "function")

**String** - a class, defined elsewhere

**method\_name:** *identifier*

convention: starts in lower case for methods

# Parameter list

**type** identifier1 , **type** identifier2 , **type** identifier3 etc.

Example:

```
static int gcd (int a, int b) // greatest common divisor
```

**type** (mandatory): any data type.

Each parameter needs its own type, even if several have the same type.

Notes:

- Parameters separated by COMMAS
- even if empty parameter list, () required.
- no semicolon after (**parameter\_list**)

Example:

```
static void moan ()  
{  
    System.out.println("This shouldn't have happened!");  
}
```

then use as:

```
moan(); // not just: moan; parentheses required
```

# Statements: declarations

- Note:**
- ANY statement terminated by SEMICOLON
  - several statements per line possible
  - statement may use more than one line of text

```
type identifier ;  
type identifier1 , identifier2 ; // same type: list with commas  
type identifier = expression; // initialization
```

## Examples:

```
int a;  
double p, q1, r_0 = 1.0;  
String s1;  
char letter = '@' ;
```

- Note:**
- the type of any variable must be declared before the variable is used.

# Statements that "do something"

Assignments: *identifier* = *expression*;

Examples: `i=0; b = c+d;`  
`i = i+1; // same as i++;`  
`s = "Hello" + ", my dear";`  
`t1 = s + ", good to see you";`

**if/else** and **while**:

**if** (*condition*) statement;

**if** (*condition*) statement1; **else** statement2;

**better: indented**

```
if (condition)
    statement1;
else
    statement2;
```

**while** (*condition*) statement;

# Expressions

## Expressions appear:

- on the right hand side of an assignment
- as an argument of a method, assigned to the method's parameter. **Example:** `square(a+1);`
- as return value for the **return** statement.

## Arithmetic expression:

- with operators `+ - * / %` [ use of blanks optional ]
- precedence: `* / %` over `+ -`. Use parentheses `()`
- **warning:** `a/b` is integer if `a,b` integer. `10/3` is 3  
fractional part of division is `a%b`, so `10 % 3` is 1

## Boolean expression (same as **condition**):

- with operators `< > == <= >=` between arithmetic expressions
- **note:** use `==` for "is equal to", `!=` for "not equal to"
- conjunction `&&` and disjunction `||` of two boolean expressions  
**warning:** evaluation will stop when result known:  
`(false && expr)`, `(true || expr)` will leave `expr` unevaluated, since result known to be **false** resp. **true**

# Grouping { } of statements

Two purposes:

- group several statements together, in particular for

```
if ( ) { } else { }
```

```
while ( ) { }
```

```
for ( ; ; ) { }
```

**Note:** no semicolon after }

- limit the **scope** of a variable (identifier):

**Example:**

```
class Test
{
    static int a = 10; // global variable of Test
    void abc() // some method
    {
        int a = 20; // this is now a new local variable
        System.out.println(a); // prints "20"
    }
}
```

# The `for ( ; ; )` statement (iterative loop)

```
for (statement1; condition; statement2)  
    statement3 ;
```

is equivalent to

```
{  
    statement1;  
    while (condition) // note: loop may never be executed  
                        // if condition false from the start  
    {  
        statement3 ;  
        statement2 ;  
    }  
}
```

Example:

```
int s=0;  
for (int i=0; i<=20; i++) // sum the first 20 integers  
    s += i ;                // add i to s
```

// note: i local to the `for` loop, now no longer available

# Special operators

**Increment operator ++ :**

`counter++ ;` is equivalent to `counter = counter + 1 ;`

**Decrement operator -- :**

`counter-- ;` is equivalent to `counter = counter - 1 ;`

**Arithmetic assignment operators += -= /= \*=**  
(add to, subtract from, divide by, multiply with)

**Examples:**

`bigsum += b;` is equivalent to `bigsum = bigsum + b;`

`position /= 2;` is equivalent to `position = position / 2;`

**Bitwise operators & |**

take the conjunction / disjunction of the bits in the integers:

`12 & 5` (binary `1100 & 0101`) is `4` , `12 | 5` is `13`

# Indenting program text

Statements belonging to the same group start in the same column (= indented by same amount). Indent after **if**, **for**, **while** .

```
When a group starts, opened with a brace "{"  
{  
    start on a new line  
    and indent by 4 blanks  
    all subsequent statements  
    and close in the same column  
    as the opening brace "{"  
    the closing brace  
    on a new line:  
}
```

**Note:** "Java in a Nutshell" moves the opening brace up {  
so that it no longer matches the closing brace  
}

This saves 1 line of vertical space but makes programs harder to read. The **vi** editor will automatically indent for you. Set tab stops to 4, and convert tabs to spaces instantly. (Configured like that in your **\_vimrc** file.)