

Java basics



Image: "Juice Fruit Juice Green Juice"
From <http://all-free-download.com/>

Public domain

Variables in Java

Unlike Python, Java is **statically typed**

So types of all variables must be declared

Local variables:

```
int i;
```

Class fields:

```
class C { boolean b; }
```

Method parameter and return values

```
public float getValue (long l) { ... }
```

Static typing in practice

Variables can only be given values that are compatible with their type

```
int i;  
i = 5;  
i = 5.0;  
i = "hello"
```

Variables cannot be redeclared with a different type

```
String s = "hello";  
int s;
```

List of Java primitive types

Type	Description	Min value	Max value	Default value
byte	8-bit signed integer value	-128	127	0
short	16-bit signed integer value	-32,768	32,767	0
int	32-bit signed integer value	-2^{31}	$2^{31} - 1$	0
long	64-bit signed integer value	-2^{63}	$2^{63} - 1$	0L
float	32-bit single precision IEEE 754 floating point value	2^{-149}	$(2-2^{-23}) \cdot 2^{127}$	+0.0F
double	64-bit double precision IEEE 754 floating point value	2^{-1074}	$(2-2^{-52}) \cdot 2^{1023}$	+0.0
boolean	Boolean value (true/false)	n/a	n/a	false
char	16-bit Unicode character value (use single quotes – e.g., 'c')	\u0000 (0)	\uffff (65,535)	\u0000

What about strings?

Technically, Java strings are of type `java.lang.String` which is an object (not primitive)

However, the Java language has special support for strings

E.g., you can create a new one by using double quotes

```
String s = "This is a string";
```

Also, they are **immutable** (will be explained later in the course)

So you will probably often tend to think of them as primitive – just don't forget that they are not!

Useful operators for primitive types

Assignment = (single equals sign)

```
int i = 5;
```

Equality comparison == (two equals signs), != (not equal to)

```
if (i == 5) { ... }
```

```
if (j != 5) { ... }
```

Relational operators <, >, <=, >=

Boolean combinations && (and), || (or)

*Special considerations for String (because it's an **object**) – stay tuned*

Type conversions

Remember, Java is a **statically typed** language

Any variable can only hold values of a single, specific type

To store a value of type t_1 in a variable of type t_2 , the value must be **converted to t_2 before** the assignment occurs

This is unlike Python where you can do something like this (and more!)

```
foo = 5  
foo = True  
foo = "bar"  
foo = 0.5
```

Implicit (widening) conversions

Sometimes, type conversion can happen automatically with no extra source code

Generally, these are *widening* conversions – little or no information lost

byte to long

long to double *// potential loss of precision*

...

Precision may be lost, but the **magnitude** of the numeric value is preserved

```
int i = 5000;
```

```
long l = i;
```

```
double d = l;
```


Explicit (narrowing) conversions – casting

Some conversions would result in significant potential information loss

`double to float`

`int to short`

...

These **narrowing** conversions must be made explicit in source code using *casting*: specify the target type in round brackets

```
int i = 1025;
```

```
byte b = (byte)i;
```

```
// b now has value 1
```

How does narrowing work?

`double` to `float`: loss of precision, plus ...

- Out-of-range values become infinite

- Some non-zero values may become zero

`double/float` to `long/int/short/byte/char`

- Round-to-zero

- Out-of-range becomes infinity

Integer type to “smaller” integer type (e.g., `long` to `int`, `int` to `byte`, ...)

- Discard all but n lowest-order bits

- In practice: value mod max

String ↔ primitive types

Converting from String

Use `parseXXX` methods of primitive wrapper classes

E.g., `int i = Integer.parseInt("42");`

Converting to String:

1. Just concatenate (+) with an existing String value
2. Use `String.valueOf(...)`

A note on integer division

Most mathematical operators (+, -, *, ...) work as you would expect in Java

Function of division operator “/” depends on type of the two arguments

If **both** are integers (`int`, `long`, `short`, `byte`, `char`), then it does **integer division**

If **either** is floating-point (`float`, `double`), then it does **floating point division**

Example:

`7.0 / 4.0` returns **1.75** (same result for `7.0 / 4` and `7 / 4.0`)

`7 / 4` returns **1**

General rule: integer division throws away the remainder (so `99 / 100 == 0`)

Blocks and scope

Block of statements is enclosed in curly braces { }

Usually all statements in a block have same indentation level

Unlike Python, this is not mandatory – but strongly recommended!

A variable is in scope (accessible) ...

From the point of declaration ...

To the end of the enclosing block

AND NOT AFTER!

```
int i = 5;
int j = 10;

{
    int i = 20;
    j = 3;
    int k = i + j;

    System.out.println(i);
    System.out.println(j);
    System.out.println(k);
}

System.out.println(i);
System.out.println(j);
System.out.println(k);
```

If statements in Java

Syntax is straightforward:

```
if (condition1)  
    block1
```

```
else if (condition2)  
    block2
```

```
// ...
```

```
else  
    blockN
```

The Java *for* loop

Syntax is similar to C, C++, C#, JavaScript, ...

Contains three clauses

Initialisation

Termination condition check

Variable update

Termination condition is checked before loop is executed

Loop variable is local to the loop – can't be used before or after

It's a normal variable though and can be updated within the loop!

```
for (int i = 0; i < 10; i++) {  
    doSomething();  
}
```

The *while* loop

Similar to – but more general than
– the *for* loop

Termination condition is checked
each time before the loop is run

- What if ***condition*** is false at the start?
- What if ***condition*** is never false?

Loop variable is available before and
after the loop (if you do a *for*-like
loop)

```
while (condition) {  
    doSomething();  
}
```

```
int i = 0;  
while (i<10) {  
    doSomething();  
    i++;  
}
```


Advanced loop control flow

for and *while* loops both have **termination conditions**

```
for (...; ...; i<10)      while (!valid)
```

Condition is checked, and if it passes, entire loop is executed before condition is checked again

But what if you want to do something different with your loop?

Skip the rest of a single loop execution – `continue`

Break out of the loop immediately – `break`

switch statements

IF STATEMENTS

Evaluates any `boolean` expression

Conditionally executes one of two statements or blocks

SWITCH STATEMENTS

Evaluates an integer or `String` expression

Integer: `byte`, `short`, `int`,
`long`, `char`, `enum`

Conditionally executes one or more of a list of case blocks

When to choose one or the other? Depends on (1) readability, (2) the nature of the expression being evaluated (complex tests require `if`)

Details of how `switch` works

Switch statement evaluates its expression ...

... and then executes **all** statements following the matching case label

... until it encounters a `break` statement

```
int vs = 0, cs = 0;
for (int i=0; i<s.length(); i++) {
    char c = s.charAt(i);
    switch (c) {
        case 'a':
        case 'e':
            // ...
            vs++;
            break;

        case 'b':
        case 'c':
            // ...
            cs++;
            break;
        default:
            System.out.println ("other");
    }
}

System.out.println("Vowels:" + vs + "Consonants:" + cs);
```

Declaring a method in Java: Syntax

A method declaration has six components (in order):

1. Access modifier(s) (zero or more – details later)
2. Return type (`void` if it does not return a value)
3. Method name (conventionally beginning with a verb)
4. Parameter list in parentheses – comma delimited list of input parameters, preceded by data type, enclosed in parens. No parameters – empty parens.
5. An exception list (more on this later)
6. The method body, enclosed in braces { }

*Method
signature*

```
public double calculateAnswer(double wingSpan, int numberOfEngines,  
                             double length, double grossTons) {  
    //do the calculation here  
}
```

Calling a method

Use the name

Specify the parameters in parentheses

Store the result in a variable of the correct type

(Not necessary if method is **void**; optional other times)

```
public double calculateAnswer(double wingSpan, int numberOfEngines,  
                             double length, double grossTons) {  
    //do the calculation here  
}  
  
double result = calculateAnswer(10, 4, 100, 1000);
```

More on methods

To return from a method, use **return** statement

Multiple **return** statements are possible!

But be sure to return the correct type

It is possible to have two methods with the same name as long as they have different parameter lists (this is called **overloading**)

```
double doSomething(int a, int b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
double doSomething(double a) {  
    return 0 - a;  
}
```

```
int doSomething(String s) {  
    return s.length();  
}
```

Declaring an array

Each array has ...

- A **type** – the type of the individual elements in the array

- A **dimension** – the dimensionality

Examples:

- `int[]` – a one-dimensional array of integers

- `String[][]` – a two-dimensional array of Strings

Variable declaration (method parameter, local variable, class field):

- `String[] args`

- `(String args[]` also valid – hang-over from C language style)

Initializing an array

Declaring an array **does not** ...

- Reserve space for the array elements

- Specify the length of the array

Technically, it only creates a **reference** to point to the array

Two options for creating an array:

- Using **new** and giving an explicit size

- Using an “initializer” to give the initial values (shortcut syntax)

Initializing an array with new

```
int[] values = new int[10];
```

Allocates enough space to store the specified number of elements

Fills the array with default values

- 0 for numeric types

- Null for object types

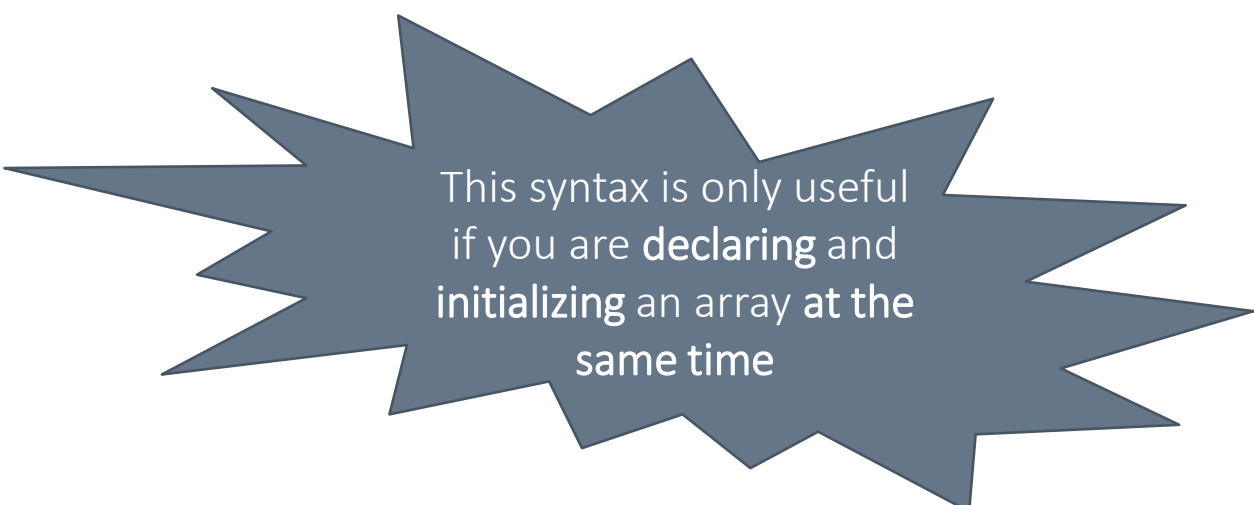
Initializing an array directly

Put a comma-separated list of elements between curly brackets

```
int[] values = { 10, 20, 30, 40, 50, };
```

Array length is determined from the length of the input list

(Nice feature: you can put a comma at the end of the list)



This syntax is only useful
if you are **declaring** and
initializing an array at the
same time

Accessing array elements

Use square brackets around index

```
int value = values[3];
```

Subscripts start at zero (as in Python – unlike other languages such as Fortran, COBOL, Matlab)

For multidimensional arrays, use multiple sets of brackets:

```
String[][] strings = // ...;  
System.out.println( strings[i][j] );
```

Array length

An array's length **cannot** be changed after it is initialized

Any attempt to use an out-of-range index will result in an error

Array length can be accessed through built-in field `length` (note: **not** a method)

```
String[] strings = { "Each", "peach", "pear", "plum", };  
System.out.println (strings.length);  
// Prints out 4
```

Iterating through an array

```
String[] fruits = new String[] { "apple", "banana", "cherry" };
```

Standard Java idiom: **for** loop

```
for (int i = 0; i < fruits.length; i++) {  
    System.out.println(fruits[i]);  
}
```

More efficient option: *for-each* loop (since Java 1.5) – similar to Python's **for** loop

```
for (String fruit : fruits) {  
    System.out.println (fruit);  
}
```