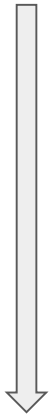
Basic Syntax

Agenda

Following topics are covered:



Java vs C#: Hello World!

Primitive variables, arrays and strings

Arithmetic, relational and logical operators

If-else, if-else-then and switch statements

Functions

Basic advices in clean coding

Hello World!

Java

```java
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World!");

    }

}
```

Variables

Fields:

```
[modifiers] type identifier [=value]
```

Local variables:

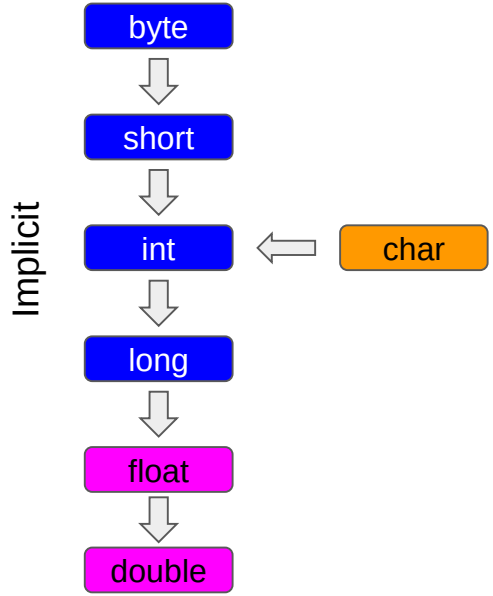
```
type identifier [=value]
```

Primitives

Primitive	Size	Default value
byte	8-bit	0
short	16-bit	0
int	32-bit	0
long	64-bit	0L
float	32-bit	0.0f
double	64-bit	0.0d
boolean	1 bit, size not defined	false
char	16-bit	'\u0000'

Implicit Casting

Implicit (narrower to wider data type) - original value is preserved

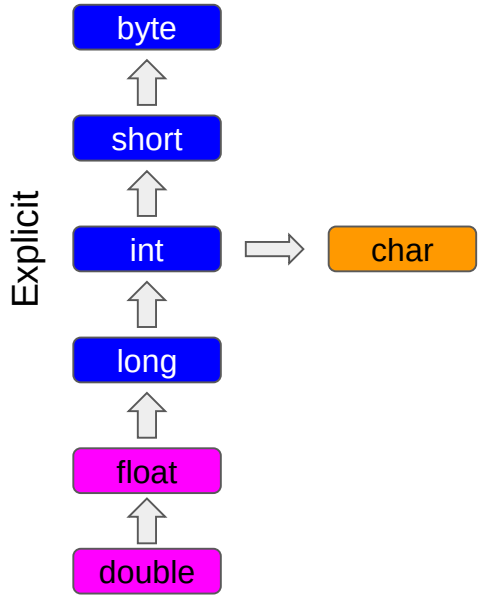



```
byte num1 = 53;

int num2 = num1;
```

Explicit Casting

Explicit (wider to narrower data type) - loss of precision




```
int num1 = 53;

byte num2 = (byte) num1;
```

Implicit / Explicit Casting

```
boolean isSkyBlue = true;

int skyBlue = isSkyBlue;
```

Question: Is cast possible?

Answer: No. Cannot cast boolean.

Variables: Naming convention

Examples:

```
public int shirtID = 1;
public String description = "-description required-";
public char colorCode = 'U';
public double price = 1e2;
public int quantityInStock = 15_000;
```

Rules:

Variable identifiers must start with either an uppercase or lowercase letter, an underscore “_”, or a dollar sign ($).

Variable identifiers cannot contain punctuation, spaces, or dashes.

Java technology **keywords** cannot be used as names.

Variables: Naming convention

Guidelines:

Begin each variable with a lowercase letter. Subsequent words should be capitalized (for example, `myVariable`).

Choose names that are mnemonic and that indicate to the casual observer the intent of the variable.

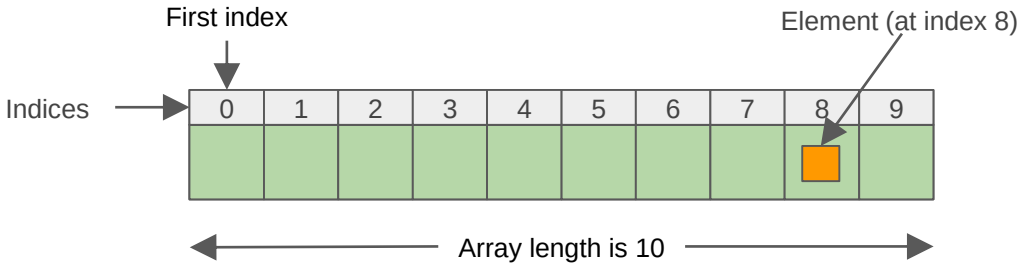
Arrays

A container object that holds a fixed number of values of a single type

Length established when created (fixed after creation)

Each item in an array is called an *element*,

Each *element* is accessed by its numerical *index*



Arrays

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

anArray[0] = 100;
```

Question: anArray[1] = **?**

Arrays

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

anArray[0] = 100;
```

Question: `anArray[1] = 0;`

Arrays

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Question: anArray[10] = **?**

Arrays

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Question: exception is thrown

Arrays

```
int[] anArray;


int anotherArray[];
```

Note: Second way of declaring an array is not by convention.

Multidimensional arrays

An array of arrays

Declaration: `type[][] name`

Access elements as in a matrix:

```
char[][] values = {
            {‘A’,’B’,’C’},
            {‘X’, ‘Y’}
        };
System.out.println(values[0][0] + “ ” + values[1][0]);
System.out.println(values[0][2] + ” ” + values[1][1]);
```

```
A X
C Y
```

Array Manipulations

Java SE provides methods to perform some of the most common manipulations related to arrays

`java.lang.System` class

- arraycopy

`java.util.Arrays` class

- copying
- sorting
- searching
- comparing

Array Manipulations

Useful operations provided by `java.util.Arrays` class:

- `binarySearch`: searching an array for a specific value to get the index
- `equals`: compare two arrays
- `fill`: fill an array to place a specific value at each index
- `sort`: sort an array into ascending order
  - sequentially, using the sort method
  - concurrently, using the `parallelSort` method introduced in Java SE 8.
    - For large arrays on multiprocessor systems is faster than sequential array sorting.

Array Manipulations

Useful operations provided by `java.util.Arrays` class:

- `asList`: convert to List type
- `copyOf`: copies to array with specified length
- `copyOfRange`: copies specified range of values from one array to another

Array Manipulations: Example

```
char[] copyFrom = { 'd', 'e', 'j', 'a', 'v', 'a', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };

char[] copyTo = new char[7];

System.arraycopy(copyFrom, 2, copyTo, 0, 4);

//copyTo holds characters:
java
```

Array Manipulations: Example

```
char[] copyFrom = { 'd', 'e', 'j', 'a', 'v', 'a', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };

char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 6);

//copyTo holds characters:
java
```

String

Sequence of characters

Are objects

The Java platform provides the String class to create and manipulate strings

```
String greeting = "Hello!";
                          OR
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '!' };
String helloString = new String(helloArray);
```

Have accessible “length” method

Individual elements can be accessed through `String` method `charAt(i)`

Concatenating Strings

When you use a string literal in Java code, it is instantiated and becomes a String reference

Concatenate strings:

```
String name1 = "Fred"
theirNames = name1 + " and " + "Anne Smith";
```

The concatenation creates a new string, and the String reference `theirNames` now points to this new string.

String is immutable, concatenating two strings **requires creating a new string**.

String Manipulations

Useful operations provided by `java.lang.String` class:

- `length()`: length of the string
- `equals`: check two string equality
- `trim`: new string with removed leading and trailing whitespaces
- `substring`: returns new string
- `indexOf`: get index of some string
- `split`: splits string into array of string by a regex
- `replaceAll`: new string with a matching regex replaced by some string

String Manipulations

Useful operations provided by `java.lang.StringBuilder` class:

- `append`: appends the argument to this string builder
- `reverse`: reverses the sequence of characters in this string builder

Note: Same as String class, but ***mutable***.

String conversion

Numerical value to string using `toString(i)` function

String to numerical values: `Type.parseType(string)`

Example:

```
double exampleValue = Double.parseDouble(textToNumExampleValue)
```

The Arithmetic Operators

+ (Addition) Adds values on either side of the operator
- (Subtraction) Subtracts right hand operand from left hand operand
*** (Multiplication)** Multiplies values on either side of the operator
/ (Division) Divides left hand operand by right hand operand
% (Modulus) Divides left hand operand by right hand operand and returns remainder
++ (Increment) Increases the value of operand by 1
-- (Decrement) Decreases the value of operand by 1

The Relational Operators

== (equal to) Checks if the values of two operands are equal or not, if yes then condition becomes true.

!= (not equal to) Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.

> (greater than) Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.

< (less than) Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.

>= (greater than or equal to) Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.

<= (less than or equal to) Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

The Logical Operators

&& (logical and) Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.

|| (logical or) Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.

! (logical not) Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

The **if-then** Statements

The `if-then` statement is the most basic of all the control flow statements.

Execute a certain section of code *only* if a particular test evaluates to `true`.

```
if(condition){
    // Do something that
    // corresponds to the condition
}
```

The **if-then** Statements

Example:

Set maximum driving speed of a vehicle

```
boolean isWithinTown = true;
int maxSpeed;
//checks if the vehicle is driving within a town area.
if (isWithinTown){
    maxSpeed = 50;//if vehicle in town, then set maximum speed to 50 km/h
}
else {
    maxSpeed = 90;
}
```

The **if-then** Statements: Ternary operator

Example:

Set maximum driving speed of a vehicle

```
boolean isWithinTown = true;
int maxSpeed = isWithinTown ? 50 : 90;
//if vehicle in town, then set maximum speed to 50 km/h, otherwise 90
```

If-then-else Statements

```
if(condition1){
    // Do something that
    // corresponds to the condition
}
else if(condition2){
    //Do something else
}
…
else if(conditionN){
    //Do something else
}
else {
    //Do something else
}
```

The switch Statement

Unlike `if-then` and `if-then-else` statements, the `switch` statement can have **any number of possible execution paths**

When *break* is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

```
switch (expression) {
    case value:
       //Statements
       break; //optional
    case value:
       //Statements
       break; //optional
    // You can have any number
     // of case statements.
    default: //Optional
       //Statements
}
```

The switch Statement

```
int expression = 2;

int result = 0;

switch (expression){
    case 1: result++;
    case 2: result++;
    case 3: result++;
       break;
    default: result++;
}
```

Question: result = ?

The switch Statement

```
int expression = 2;

int result = 0;

switch (expression){
    case 1: result++;
    case 2: result++; // Matches this case and executes any code
until a ‘break’ found
    case 3: result++;
       break;
    default: result++;
}
```

Question: result = 2;

Loop Controls: While, Do While, FOR

while

```
while (Boolean_expression) {
    // statements
}
```

do while

```
do {
   // statements
} while (Boolean_expression);
```

for

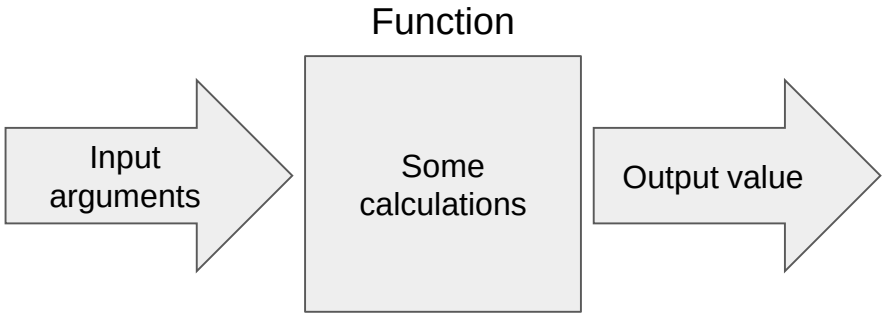
```
for (initialization; Boolean_expression; update) {
   // statements
}
```

Functions (methods in OOP context)

Named block of code containing a series of statements.

Program fragment that 'knows' how to perform a defined task.

Takes an input , does some calculations on the input, and then gives back a result.



Functions

Should be as small as possible

FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY. [Robert C. Martin, 2008, *Clean Code*]

Defining functions

Required elements of a function declaration

Return type,

Name,

A pair of parentheses for arguments, (type param1, type param2, ….), or simply ()

A body between braces, {}.

```
public double avg(double a, double b, double c){
    double result = (a + b + c) / 3;
    return result;
}
```

Arbitrary Number of Arguments

Use *varargs*

shortcut to creating an array manually

type of the last parameter + three dots '...' + parameter name

```
public Polygon polygonOfPoints(Point... cornerPoints) {
    int numberOfSides = cornerPoints.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1    = (cornerPoints[1].x - cornerPoints[0].x)
                     * (cornerPoints[1].x - cornerPoints[0].x)
                     + (cornerPoints[1].y - cornerPoints[0].y)
                     * (cornerPoints[1].y - cornerPoints[0].y);
    lengthOfSide1    = Math.sqrt(squareOfSide1);
}
```

Putting everything together: Hello World!

```
package helloworld;
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Summary: JAVA keywords

abstract	**assert**	**boolean**	**break**	**import**	instanceof	**int**	**interface**	volatile	**while**
byte	**case**	**catch**	**char**	**long**	native	**new**	**package**		
class	**const**	**continue**	**default**	**private**	**protected**	**public**	**return**		
do	**double**	**else**	**enum**	**short**	**static**	strictfp	**super**		
extends	**final**	**finally**	**float**	**switch**	**synchronized**	**this**	**throw**		
for	goto	**if**	**implements**	**throws**	transient	**try**	**void**		

Summary: Clean Code

Goal:

understand code

quality (bug-free) and efficiency (optimal use of resources)

Good variable names:

Variable names should be self-descriptive

```
int d; // elapsed time in days
```

BAD

OR

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Summary: Clean coding

Variable name VS its contained information

```
int classSize = 95;
boolean isClassSize95 = true;
```

BAD

Summary: Clean coding

Understanding and changing the code

May be hard even without complex expressions

Problem - implicitly not simplicity: the degree to which the context is not explicit in the code itself

Avoid disinformation

Avoid leaving false clues that obscure the meaning of code

Do not encode the container type into the name (for example `accountList`)

Beware of using names which vary in small ways

Example: lower-case **L** or uppercase **O** as variable names, especially in combination:

```
int a = l;
if ( O == l )
   a = O1;
else
   l = 01;
```

Summary: Clean coding

- Avoid noise words:
  - Why? They are redundant (for example, `info` and `data`).
  - The word `variable` should never appear in a variable name. The word `table` should never appear in a table name.
- Pronounceable, searchable names
- Avoid magic numbers ()
- Don’t pun: **Say what you mean! Mean what you say!**
- Develop good descriptive skills
- Write code for other person not for computer

Thank you for attention

Home reading

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html