**Complete Java Masterclass**By Tim Buchalka

**UI / Shortcuts**

* Use Code->Reformat Code to fix code formatting automatically.
* Double LMB the tab to make it full screen.
* Highlight code then press Ctrl+/ to comment it out/in.

**General Language Features**

**Section 4 – Variables, Datatypes and Operators**

To output something to the console: System.out.println(). You can type sout and press enter to generate that quickly.

A fixed value is called a **literal**, e.g. *5, 10.0, 244L, ‘a’, true, “hello”, etc*. If the statement contains values, variables, operators or method calls, it’s an **expression**. You ignore the datatype, *e.g. int num = 5; 5* is the literal, and *num = 5* is the expression. Expressions have been highlighted, and literals have been bolded:

int score = **100**;  
if (score > **99**) {

System.out.println(**“You got the high score!”**);

score = **0**;

}

**Integers** (*int*) take 4 bytes (32 bits) and can therefore store 232 possible values. Half of these are in the positive range, and half in the negative. 0 is counted as a positive value, hence the positive range is -1 of the negative range. So *int max = 2147483647*, and *int min = -2147483648*.

Literals can contain **underscores** to make them easier to read, e.g. *int max = 2\_147\_483\_647*. The underscores can be played anywhere in the sequence if it’s not in the beginning or end, e.g. *int max = 2\_\_1\_47\_483\_647*.

**Byte** (*byte*) is just 1 byte (8 bits) in size. It can only store values from -128 to 127. There isn’t anything smaller than a byte since a byte is the smallest data that can be fit into memory.

**Short** (*short*) is 2 bytes (16 bits) in size. It can store values from -32768 to 32767.

If the literal value exceeds these limits, then the compiler will give an error. You can cast the value to ‘wrap around’, e.g. *byte num = (byte) 128* (num will be -128), *byte num = (byte) 129* (num will be -127), etc.

**Width** is the size of a datatype in bits, e.g. width of short is 16.

**Long** (long) is 8 bytes (64 bits) in size. To create a long literal you must append L, e.g. *long num = 100L*. You can use lowercase L, but uppercase is easier to read.

In width order: byte < short < int < long. To convert a larger width datatype to a smaller width datatype you will have to cast. So all other types must be manually casted to byte, whereas none of them need to be casted to long as the compiler will automatically do it. Examples: *int num1 = 5*; byte *num2 = (byte) num1*; *long num3 = num1*.

**Float** (float) is 4 bytes in size. To create a float literal, you must append f or F. A float has 7 digits of precision. You can use underscores with floating point values. If you attempt to do calculations with the previous types that include decimals, the decimal/remainder will be disregarded, e.g. *int num = 5/2* will return 2 since .5 is ignored. Float and double keep the remainder.

**Double** (double) is 8 bytes in size. To create a double literal, you must append d or D, or type a decimal value, e.g. *double num = 5d*, *double num = 5.25d*, or *double num = 5.25*. A double has 16 digits of precision. Double calculations tend to be faster than float calculations on modern computers, even though it takes more space.

In width order: float < double. As previously stated, larger types must be manually casted to small types. Smaller types automatically cast to larger types.

**Char** (char) takes 2 bytes. It’s used to store a single character such as a letter, number, Unicode character, or escape sequence. Example: *char val = ‘A’*, or *char val = ‘\u00A9’* (Unicode for copyright symbol). The apostrophes define a character literal. Unicode character tables may be useful: <https://unicode-table.com/en/>.

**Boolean** (boolean) takes 1 byte. It stores the state of something, e.g. is it true or false, or *boolean isMale = true*. ‘true’ and ‘false’ are Boolean literals.

All the datatypes covered up to now are primitive types defined within Java. These basic datatypes can be used to create more sophisticated datatypes, such as string. All other datatypes are referred to as a class.

**Strings** (String) takes a variable number of bytes. It is a sequence of characters, so it will take 2 bytes for each character you type (including spaces), plus another 2 bytes since strings are automatically null terminated. Speech marks are used to define a string literal. Example: *String str = “Hello World”* will take 24 bytes. Strings support direct concatenation, e.g. *String str = “Hello” + “ world!”*. It can also do this with all primitive types, e.g. *int num = 50; String str = “Hello ” + num*, output: “Hello 50”.

**Operators** can work on 1 (unary), 2 (binary) or 3 (ternary) operands. Unary ops: (post-fix, pre-fix). Binary ops: arithmetic (multiplicative, additive), assignment (=, +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=), bitwise (<<, >>, >>>, &, ^, |), logical (&&, ||), relational (comparison, equality). Ternary operator: conditional operator.

When assigning values, you can chain them, e.g. *int a, b, c; a = b = c = 5;*. This is read from right to left and sets all to 5. First c is set to 5, then b is set to c, then a is set to b.

**Relational (comparison, equality) and logical operators** check to see if the two operands match a condition, in which case the output is true, otherwise false. All of them are binary operators except for the not operator, which is unary.

The **conditional operator** checks to see if the condition is true, and returns the first statement, otherwise it returns the other statement, e.g. *(5>4) ? “Yes” : “No”;* returns “Yes”. In general the syntax is *condition ? if\_true : if\_false;*. You can return literals, expressions, statements, etc.

For an if statement to be valid, the expression must evaluate to a boolean. This is why *boolean bool = false; if (bool = true);* compiles, but *int num = 5; if (num = 5);* doesn’t.

**Section 5 – Expressions, Statements, Code Blocks, Methods, etc.**

* There are 53 reserved keywords in Java.
* **Scope** refers to the **block** {} in which a variable is defined. Variables are automatically deleted once they fall out of scope - at the end of the block they were defined in. Variables cannot be accessed outside of their scope.
* **Methods** allow you to type code once and reuse it as needed, thus preventing code duplication.
* **Parameters** are datatypes + variable names expected by a method. **Arguments** are the actual values/objects being supplied to a method call.
* **Procedures** don’t return anything. **Functions** return something. **Methods** are procedures or functions defined within a class.
* **Method overloading** is when a new method is created that has the same name as another method, but has different parameters. The compiler decides which method to call based on the arguments provided.

**Section 6 – Control Flow Statements**

* These change the flow of the code depending on various conditions, e.g. **if statement, switch statement, for loop, while loop, do while loop.**
* If statements are like switch statements. Switch statements can only test one variable, can only test for equality (can be OR’ed), and can only equate constant values. Switch statements should only be used for simple test cases. Examples:

|  |  |
| --- | --- |
| int num = 3; if (num == 1)  System.out.println(“1”);  else if (num == 2 || num == 3)  System.out.println(“2 or 3”);  else  System.out.println(“<1 or >3”); | int num = 3;  switch (num) {  case 1:  System.out.println(“1”);  break;  case 2: case 3:  System.out.println(“2 or 3”);  break;  default:  System.out.println(“<1 or >3”);  break;  } |

* An if statement’s ‘else if’ is equivalent to a switch statement’s ‘case’. An if statement’s ‘else’ is equivalent to a switch statement’s ‘default’.
* You must type break after every case or execution will fall through to the next case, this is how you OR different cases together.
* Switch statements can only switch on byte, short, int, char, their wrapper classes, enum, and String.
* Syntax for an indexed for loop is: *for (init; termination; increment) {}*.

|  |  |
| --- | --- |
| for (int i = 0; i < 5; i++) {  System.out.println(i);  } | for (int i = 0, j = 5; i < 5; i++, j--) {  System.out.println(i + " " + j);  } |

* By convention, simple variables are used to track the index, e.g. *int i, int j, int k, …*
* *String.format("First: %d Second: %.2f Third: %c", 5, 10.0, 'c')*, will output “First: 5 Second: 10.00 Third: c”. The % notation refers to the corresponding argument, e.g. the first % refers to the first argument etc.
* Nested if statements are more efficient than sequential if statements, if the sequential if statement depends on code executed in the former if statement. Example:

|  |  |
| --- | --- |
| int primes = 0;  for (int i = 1; i < 20; i++) {  if (isPrime(i)) {  System.out.println(i);  primes++;  }    if (primes >= 3)  break;  }  } | int primes = 0;  for (int i = 1; i < 20; i++) {  if (isPrime(i)) {  System.out.println(i);  primes++;  if (primes >= 3)  break;  }  } |

The code on the left checks two conditions every time, while the code on the right checks two conditions sometimes.

* For loops are ideal for looping a certain number of times. While loops are ideal for looping until a certain condition is met. Do while loops are the same as while loops, but run at least once since the condition is checked after the loop.
* You can use the **break** keyword to exit a loop if a condition is met, or you can use the **continue** keyword to start the next iteration of the loop.

**Section 7 – OOP Part 1: Classes, Constructors, and Inheritance**

* asdasd