Week 03 - Material

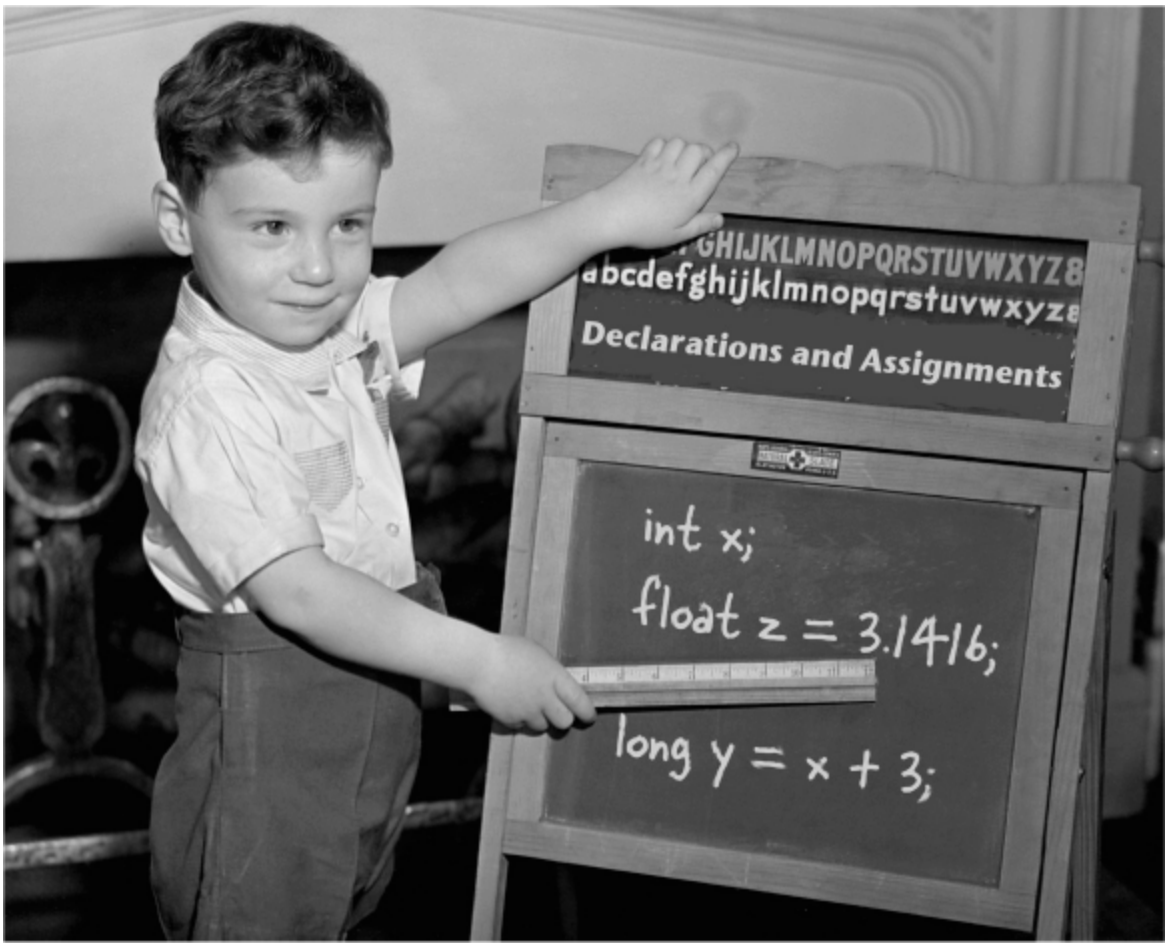
Louis Botha, `louis.botha@tuni.fi`

Variables, User Input, Conditional Logic and Loops

What should Java developers learn in 2021? - DataScienceCentral.com

Next we will look at the basic building blocks of our programming languages. We are looking at Java but the fundamental ideas are exactly the same independent of programming language. This weeks lecture and exercises are the base for all your future programming.

Variables



Head First Java

When declaring a variable you must follow this two declaration rules:

> ***variables must have a type***
> ***variables must have a name***

A variable needs a type and it needs a name, so that you can use that name in code.

Variable Type and Name

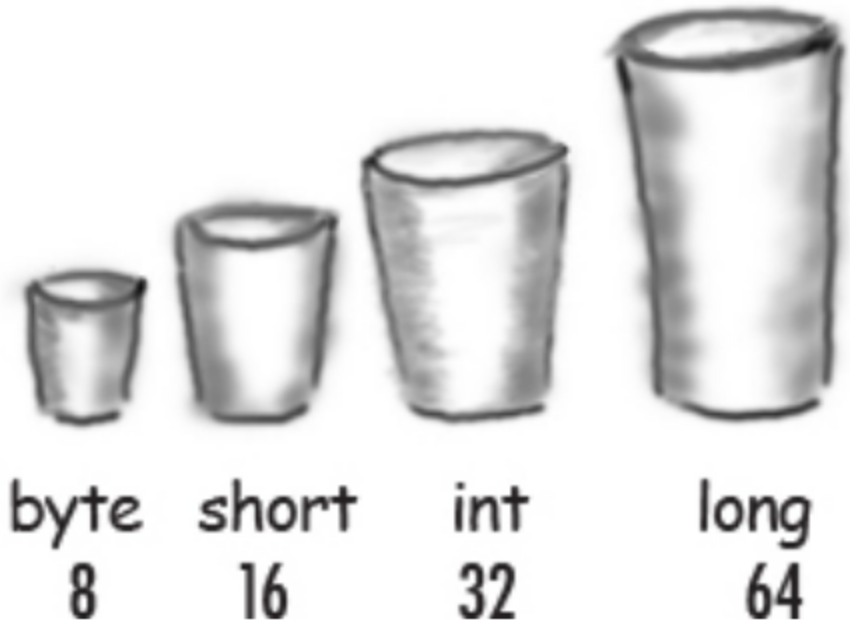
Variables come in two flavours: **primitive** and **object reference**.

Primitives hold fundamental values (think: simple) including integers, booleans, and floating point numbers. Object references hold, well, references to objects (We will get to objects later.)

The eight primitives defined in Java are *int*, *byte*, *short*, *long*, *float*, *double*, *boolean* and *char*. These aren't considered objects and represent raw values.

Type	Size (bits)	Minimum	Maximum	Example use
byte	8	-2^7	$2^7 - 1$	*byte b = 100;*
short	16	-2^{15}	$2^{15} - 1$	*short s = 30_000;*
int	32	-2^{31}	$2^{31} - 1$	*int i = 100_000_000;*
long	64	-2^{63}	$2^{63} - 1$	*long l = 100_000_000_000_000;*
float	32	-2^{-149}	$(2-2^{-23})\cdot 2^{127}$	*float f = 1.456f;*
double	64	-2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	*double f = 1.456789012345678;*
char	16	0	$2^{16} - 1$	*char c = 'c';*
boolean	1	–	–	*boolean b = true;*

> *:protip: Think of a variable as just a cup. A container. It holds something. It has a size, and a type.*



Head First Java

> *:attention: The memory size the types need to store a value is "nice to know" to stuff for those that are interested. The key takeaway is that when the memory use increase, so increase the size of the number.*

int

Also known as an integer, *int* type holds a wide range of non-fractional number values and **it uses 32 bits of memory.** It can represent values from -2,147,483,648 (-2^{31}) to 2,147,483,647 (2^{31}-1).

We can simply declare an *int*:

```
int x = 424000;
int y;
int z = -4;
```

- The default value of an *int* declared without an assignment is 0.
- If the variable is defined in a method, we must assign a value before we can use it.

byte

byte is a primitive data type similar to *int*, except **it only takes up 8 bits of memory.** This is why we call it a byte. Because the memory size is so small, *byte* can only hold the values from -128 (-2^7) to 127 ($2^7 - 1$).

Here's how we can create *byte*:

```
byte b = 100;
byte empty;
```

- The default value of *byte* is also 0.

short

The next stop on our list of primitive data types in Java is *short*. If we want to save memory and *byte* is too small, we can use the type halfway between *byte* and *int* called *short*.
At 16 bits of memory, it's half the size of *int* and twice the size of *byte*. Its range of possible values is -32,768(-2^{15}) to 32,767($2^{15} - 1$).

short is declared like this:

```
short s = 20020;
short s;
```

- The default value is 0.
- We can use all standard arithmetic on it as well.

long

Our last primitive data type related to integers is *long*.
long is the big brother of *int*. **It's stored in 64 bits of memory**, so it can hold a significantly larger set of possible values.
The possible values of a long are between -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63} - 1$).

We can simply declare one:

```
long l = 1234567890;
long l;
```

- The default value is also 0.

- We can use all arithmetic on *long* that works on *int*.

float

We represent basic fractional numbers in Java using the *float* type. This is a single-precision decimal number. This means that if we get past six decimal points, the number becomes less precise and more of an estimate. **This type is stored in 32 bits of memory just like *int*.** However, because of the floating decimal point, its range is much different. It can represent both positive and negative numbers. The smallest decimal is $1.40239846 \times 10^{-45}$, and the largest value is $3.40282347 \times 10^{38}$.

We declare *float*s the same as any other type:

```
float f = 3.145f;

float f;
```

- And the default value is 0.0 instead of 0.
- Notice we add the *f* designation to the end of the literal number to define a float. Otherwise, Java will throw an error because the default type of a decimal value is *double*.

double

The name comes from the fact that it's a double-precision decimal number. **It's stored in 64 bits of memory.** This means it represents a much larger range of possible numbers than *float*. The range can also be positive or negative.

Declaring *double* is the same as other numeric types:

```
double d = 3.13457599923384753929348D;

double d;
```

- The default value is also 0.0 as it is with *float*.
- We attach the letter *D* to designate the literal as a double.

boolean

The simplest primitive data type is *boolean*. It can contain only two values: *true* or *false*. **It stores its value in a single bit.** *boolean* is the cornerstone of controlling our programs flow.

Here's how we declare *boolean*:

```
boolean b = true;

boolean b;
```

- Declaring it without a value defaults to *false*.

char

The final primitive data type to look at is *char*. Also called a character, *char* is a 16-bit integer representing a Unicode-encoded character. Its range is from 0 to 65,535. In Unicode, this represents *'\u0000'* to *'\uffff'*. For a list of all possible Unicode values, check out sites such as Unicode Table.

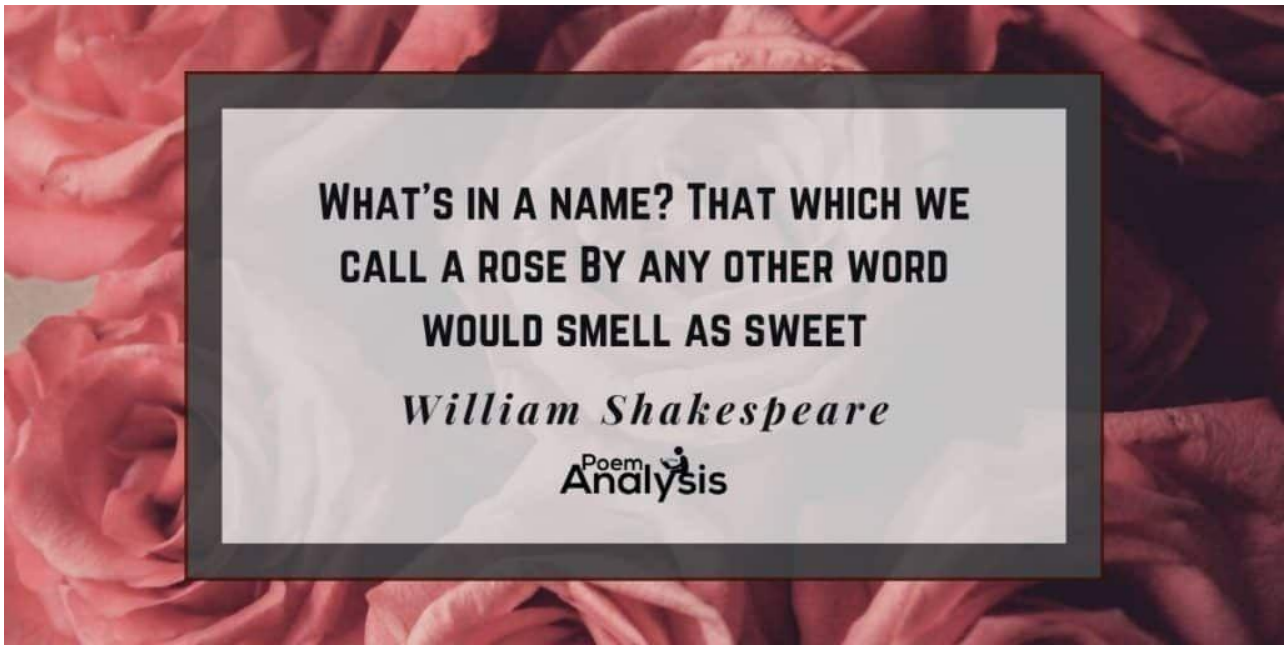
To declare a *char*:

```
char c = 'a';

char c = 65;

char c;
```

- When defining our variables, we can use any character literal, and they will get automatically transformed into their Unicode encoding for us. A character's default value is *'/u0000'*.

Variable Names

What can we name our variables?



A rose by any other name would smell as sweet meaning

The rules are simple. You can name a class, method, or variable according to the following rules (the real rules are slightly more flexible, but these will keep you safe):

- **It must start with a letter, underscore (_), or dollar sign ($).**
- **You can't start a variable name with a number.**
- **After the first character, you can use numbers as well. Just don't start it with a number.**
- **It can't be one of Java's reserved words.**
- **use camel case.**

Naming variables is a fundamental aspect of describing a program. Let's look at two examples of code that is doing exactly the same thing, one of them is, however, much more understandable.

```
double a = 3.14;
double b = 22.0;
double c = a * b * b;
System.out.println(c);

// Sample output
// 1519.76
```

```
double pi = 3.14;
double radius = 22.0;
double surfaceArea = pi * radius * radius;
System.out.println(surfaceArea);

// Sample output
// 1519.76
```

Reading input

To make the exercises a bit more fun and interactive, we will need to read some input from the user. Input refers to text written by the user and read by the program. For reading input, we use the `Scanner` tool that comes with Java. The tool can be imported for use in a program by adding the command `import java.util.Scanner;` before the beginning of the main program's frame (`public class Program {`).

The tool itself is created with:
`Scanner scanner = new Scanner(System.in);`

```
import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // We can now use the scanner tool.
        // This tool is used to read input.
    }
}
```

```
// Introduce the scanner tool used for reading user input
import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        // Create a tool for reading user input and name it sc
anner
        Scanner scanner = new Scanner(System.in);

        // Print "Write a message: " to tell the user we are e
xpecting some input
        System.out.println("Write a message: ");

        // Read the string written by the user, and assign it
        // to a variable
        // String message = (string that was given as input)
        String message = scanner.nextLine();

        // Print the message written by the user
        System.out.println(message);
    }
}
```

Input is always read as a string when using `nextLine()` but we can tell the scanner tool to expect another type.

Method	Description
`nextShort()`	reads a `short` value from the user
`nextLong()`	reads a `long` value from the user
`nextLine()`	reads a line of text from the user
`nextInt()`	reads an `int` value from the user

`nextFloat()`	reads a `float` value form the user
`nextDouble()`	reads a `double` value from the user
`nextByte()`	reads a `byte` value from the user
`nextBoolean()`	reads a `boolean` value from the user
`next()`	reads a word from the user

What do you need to remember when you use for example the `nextInt()` method to read the input?
You need to store it in an int variable of course `int x = scanner.nextInt();`

Conditional Logic Concepts

Conditional logic helps to control the flow of a code in Java. It consists of the following components:

- Test expression(s)
- Body/code block to be executed

Operators

Logical operators

- && AND operator
- ! NOT operator
- || OR operator
- Order of evaluation - Parenthesis, NOT, AND, OR

Examples:

```
boolean x = true || false;
boolean y = false && true;

System.out.println(!x);

// Output
```

```
true
```

Conditional operators

- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to
- == equal to
- != not equal to

Complete the table by writing in each cell if the result of the condition is **true** or **false.**

number	number > 0	number < 10	number > 0 && number < 10	!(number > 0 && number < 10)	number > 0 || number < 10
-1					
0					
1					
9					
10					

w3schools: Java operators

if statement

- Evaluates test expression given in parenthesis
- If test expression evaluates to `true`, then the statements in the code block are executed

Syntax:

```
if(condition) {
  statements..
}
```

Example:

```
// Using modulo % to print if number is odd or even
if(number % 2 == 0) {
  System.out.println("number is even");
}
```

if-else statement

- Evaluates test expression given in parenthesis
- If test expression evaluates to `true`, then the statements in the code block are executed
- If test expression evaluates to `false`, then the statements under the `else` block are executed

Syntax:

```
if(condition) {
  statements..
} else {
  statements..
}
```

Example:

```
// Using modulo % to print if number is odd or even
if (number % 2 == 0) {
  System.out.println("number is even");
} else {
  System.out.println("number is odd");
}
```

Handling Multiple Conditions

> *:attention: Chaining and nesting can make the code difficult to read*

if... else if... else

Syntax:

```
if(condition1) {
  statements..
} else if(condition2){
  statements..
} else {
  statements..
}
```

Example:

```
// Print if number is odd or even
if(number == 0) {
  System.out.println("number is 0");
} else if(number % 2 == 0) {
  System.out.println("number is even");
} else {
  System.out.println("number is odd");
}
```

Three or more *if/else* statements can be hard to read. As one of the possible workarounds, we can use *switch,* as seen below.

switch

If we have multiple cases to choose from, we can use a *switch* statement.

Let's again see a simple example:

```
int count = 3;
switch (count) {
case 0:
    System.out.println("Count is equal to 0");
    break;
```

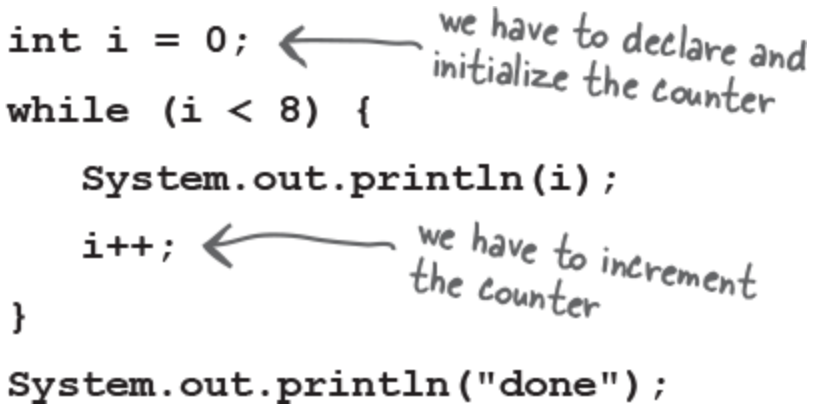
```
case 1:
    System.out.println("Count is equal to 1");
    break;
default:
    System.out.println("Count is either negative, or higher th
an 1");
    break;
}
```

Loops

We use loops when we need to repeat the same code multiple times in succession.

While

A *while loop* is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.



Head First Java

```
int i = 1;
while (i < 8) {
```

```
  System.out.println(i);
  i++;
}

//Output
0
1
2
3
4
5
6
7
done
```

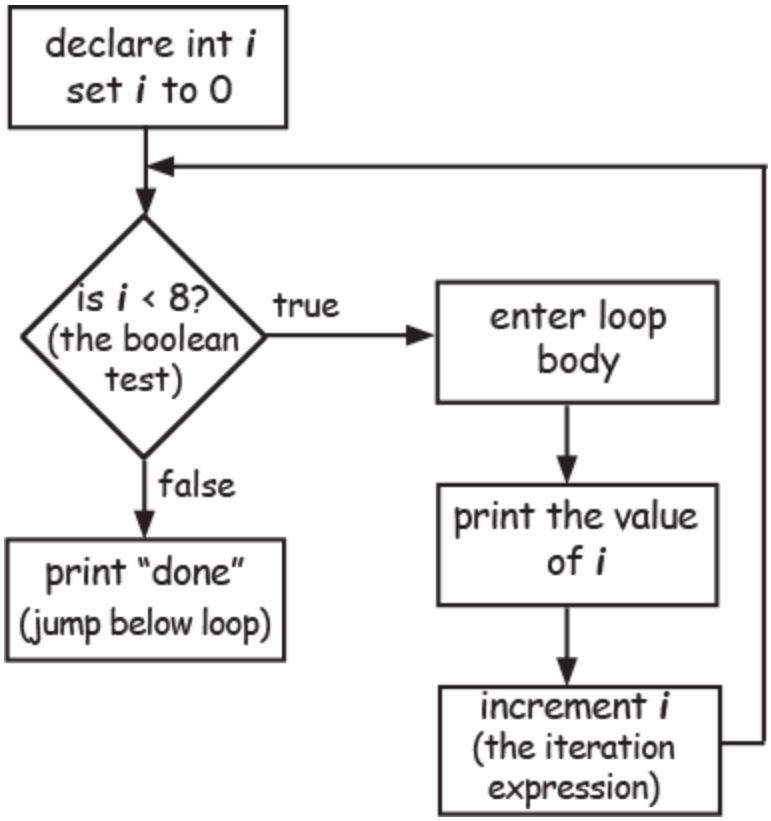
Always remember to check/verify that your while condition is changing. If your condition never becomes false then you have an infinite loop.

```
while(true) {
    System.out.println("I can program!");
}
```

You will have to end this code with CTRL + C.

For

A *for loop* is used to iterate a part of the program certain amount of times. On paper it looks like this:



Head First Java

In the Java programming language you implement a for loop like this:

```
for(int i=0; i<=8; i++) {
    System.out.println(i);
}
System.out.println("done");
//Output
0
1
2
3
4
5
6
7
```

```
done
```

Break

We can use *break* to exit early from a loop. Let's see a quick example:

```
for(int i=1; i<=7; i++) {
    if(i == 5) {
        break;
    }
    System.out.println(i);
}


1
2
3
4
```

Here, when i is 5, we want to stop the loop. A loop would normally go to completion, but we've used *break* here to short-circuit that and exit early.

Continue

Simply put, *continue* means to skip the rest of the loop we're in:

```
for(int i=1; i<=7; i++) {
    if(i == 5) {
        continue;
    }
    System.out.println(i);
}


1
2
```

```
3
4
6
7
```

Here, when i is 5, we skip printing i to the screen and continue from 6.