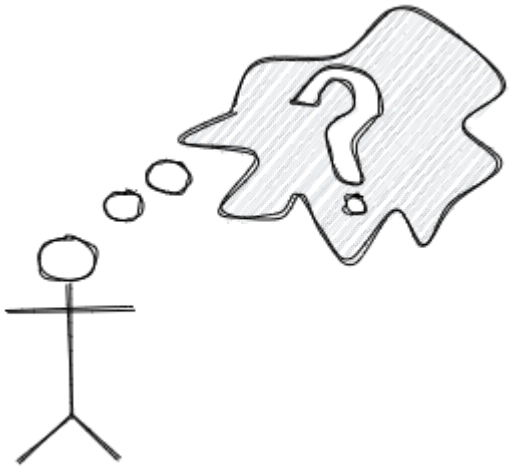
Java Fundamentals Course - Day 2

Looking Back

- Introduction to Programming
- Introducing Java
- Variables & Types

Introduction to Programming



- Who remembers …
    - what is programming?
    - what is abstraction?

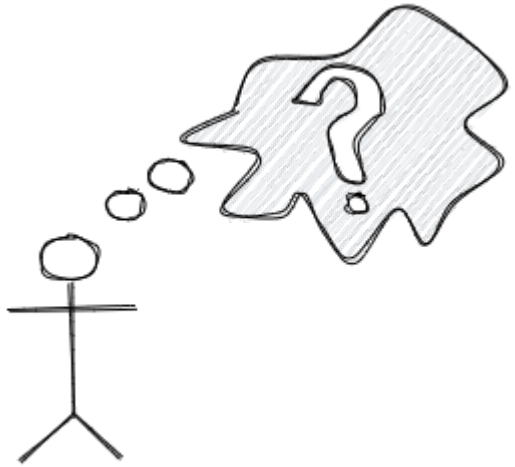
What is Programming?

- A Program is a set of instructions that a computer can run
- A Programmer is a person solving problems by writing Programs

What is Abstraction?

- You do not need to know *all* the details
- Pick the parts you need for your problem
- Chunks a problem down to its root

Introduction to Java



- Who remembers …

    - Java’s tagline?
    - some of Java’s features?
    - the difference between compile-time and runtime errors?

Features of Java

- Write Once, Run Everywhere
- Java Virtual Machine
- Strong Typing
- Garbage Collection / Memory Management
- Object Orientation

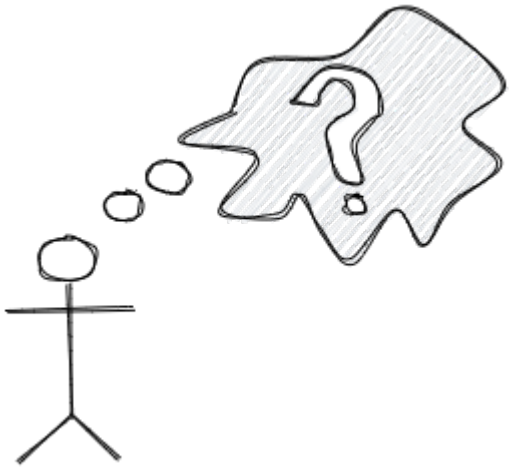
Compile-time vs runtime errors

- Compile time errors
    - Incorrect usage of syntax
    - Spelling errors in names
    - Type errors
- Runtime errors
    - Mistakes in conditions
    - Invalid input from users

- Unexpected situations

Larger Projects

Variables & Types



- Who remembers …

Module: Decisions and Logic

Boolean Algebra

- Boolean Algebra is a mathematical system that uses:
  - Two values, denoted by `1/0` or `yes/no` or `true/false` or `on/off` or `high/low` or ¬/¬ etc.

    - Three operations: `AND`, `OR`, `NOT`
- Every logical formula can be expressed using these operators
- A boolean value can be represented using a single bit.
- It is fundamental for practically all digital devices.

Primitive Boolean Type

> The `boolean` type represents a logical quantity with two possible values, indicated by the literals `true` and `false`.
>
> — The Java® Language Specification, Java SE 17 Edition, §4.2.5: The boolean Type and boolean Values

Example for boolean variables.

```
boolean sunShines = true;
boolean itRains = false;
```

Boolean Operators

- Using only `AND`, `OR`, `NOT` would not be very convenient.
- Important values and operators:
    - Constants:
      `true`, **`false`** (or `0`, `1`…)
    - Unary operators:
      `NOT`, identity, always `true`, always `false`

- Binary operators:

  AND, **OR**, **XOR**, NAND, NOR, XNOR, IMPLY, equality, …

Boolean Operators: NOT

NOT x = { true - if x is false
        { false - if x is true

Java Example for logical NOT

```
boolean thereAreClouds = true;
boolean skyIsClear = !thereAreClouds;
```

Boolean Operators: AND

x AND y = { true - if both x and y are true
          { false - otherwise

Java Example for logical AND

```
boolean sunShines = true;
boolean warmTemperature = true;
boolean weatherIsNice = sunShines && warmTemperature;
```

Boolean Operators: OR

x OR y = { true - if either x or y is true
         { false - otherwise

Java Example for logical OR

```
boolean itRains = false;
boolean coldTemperature = true;
boolean weatherIsBad = itRains || coldTemperature;
```

Boolean Operators: XOR

x XOR y = { true - if x and y are not equal
false - otherwise not eq...

Java Example for logical XOR

```
boolean itRains = false;
boolean umbrellaClosed = true;
boolean umbrellaUsedCorrectly = itRains ^ umbrellaClosed;
```

Truth Tables

- Instead of writing a (semi-)formal definition of a boolean function, we can also simply list all possible cases.
- Idea: Summarize all inputs and outputs in a **truth table**.
  - The first columns contain the values for the input variables.
  - The next column contains the result of the boolean operation.
    - It can be useful to put multiple functions in a single table.
    - For complex expressions we can add multiple columns to evaluate sub terms.

    - Each row corresponds to a unique input combination.
- For each boolean input variable there are only two possible values.
    - So for n variables we need 2^n rows.

Truth Table: NOT

A	!A
false	true
true	false

Truth Table: AND, OR, XOR

A	B	A && B	A || B	A ^ B
false	**false**	false		
false	**true**	false		
true	**false**	false		
true	**true**	true		

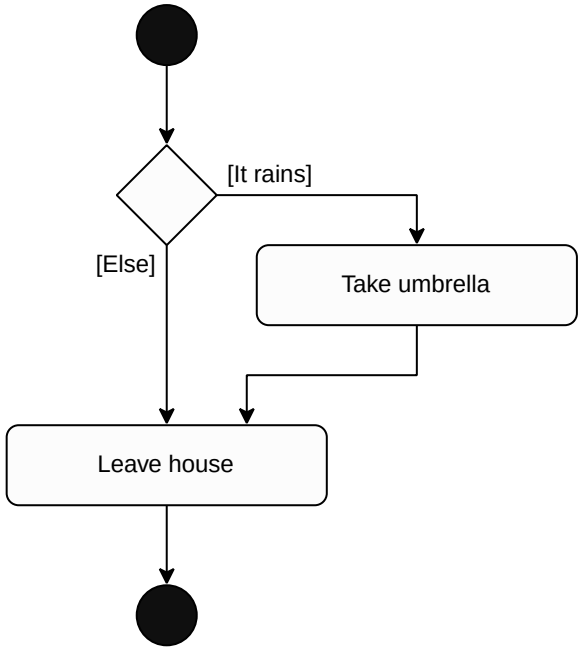
Truth Table: AND, OR, XOR

A	B	A && B	A || B	A ^ B
false	**false**	false	false	false

A	B	A && B	A || B	A ^ B
false	**true**	false	true	true
true	**false**	false	true	true
true	**true**	true	true	false

if-then

- Programs are rarely just a fixed sequence of instructions.
- `if` statements allow us to execute parts of the code only if some conditions are met.
- Example: If it rains, take an umbrella with you when leaving the house.



if-then (Java Example)

```
class IfThenElseDemo{
    public static void main(final String[] args){
        final boolean randomBoolean = Math.random() > 0.5;   // 50% chance for `true`
        System.out.println("Hello");
        if(randomBoolean) {
            System.out.println("Surprise!");
        }
        System.out.println("Goodbye");
    }
}
```

Sample Output A

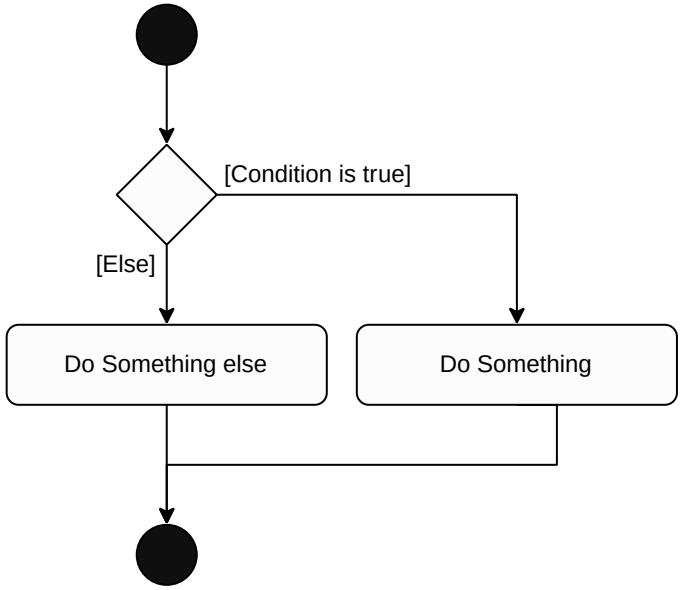
```
Hello
Surprise!
Goodbye
```

Sample Output B

```
Hello
Goodbye
```

if-then-else

- Sometimes you want to choose between two different actions depending on some condition.
- This can also be done with `if` statements.



if-then-else (Java Example)

```
class IfThenElseDemo{
    public static void main(final String[] args){
        System.out.println("Flipping coin...");
        final boolean randomBoolean = Math.random() > 0.5;   // 50% chance for `true`
        if(randomBoolean) {
            System.out.println("Heads!");
        }
        else {
            System.out.println("Tails!");
        }
        System.out.println("Goodbye!");
    }
}
```

Sample Output A

```
Flipping coin...
Heads!
Goodbye!
```

Sample Output B

```
Flipping coin...
Tails!
Goodbye!
```

Comparing Primitive Values

- Java offers operators for comparing primitive tpyes:
    - Numerical comparison:
        - < (less)
        - <= (less or equal)
        - >= (greater or equal)
        - > (greater)
    - Equality:
        - == (equal)
        - != (not equal)
- The result of a comparison is always a boolean value.

- **WARNING**:
  Equality comparison using `==` and `!=` only works for primitive types in Java. These operators can also be applied on objects, but have a different effect. You'll learn more about this later in the course.

Integral Comparison

- Works with all integral types: `byte`, `short`, `int`, `long`, `char`
- These operators can be used to compare whole numbers: <,<=,==,!=,>=,>

```
System.out.println(  1  ==  1); // prints "true"
System.out.println(  1l ==  1); // prints "true"
System.out.println(  1  !=  2); // prints "true"
System.out.println( -5  >  -1); // prints "false"
System.out.println(  0  <=  1); // prints "true"
System.out.println( '1' ==  1); // prints "false"
```

Floating-Point Comparison

- Works with all floating-point types: `float`, `double`
- It is also possible to compare integral types to floating-point types. They will then be treated as floating-point values, too.
- Similar to integral types we can use <,<=,==,!=,>=,>.
- However, there are some caveats:
  - Special values: `NaN`, `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`
  - Ambiguous values: `-0.0`, `0.0`

- "Fuzziness" of floating-point values

Floating-Point Comparison: Special values

- `NaN` is not less, equal, or greater than anything (including itself)
- `NEGATIVE_INFINITY` is less than any number.
- `POSITIVE_INFINITY` is greater than any number.

```
System.out.println( NaN ==  NaN); // prints "false"
System.out.println( NaN >   0.0); // prints "false"
System.out.println( 0.0 > NEGATIVE_INFINITY); // prints "true"
System.out.println( 0.0 < POSITIVE_INFINITY); // prints "true"
```

Floating-Point Comparison: Ambiguous values

- There are multiple different bit pattern for `NaN`. They all behave the same w.r.t. comparison in Java.
- `-0.0` and `0.0` are different values, but they are considered equivalent when being compared.

```
System.out.println(-0.0 >  0.0); // prints "false"
System.out.println(-0.0 <  0.0); // prints "false"
System.out.println(-0.0 == 0.0); // prints "true"
```

Floating-Point Comparison: Fuzziness

- The floating-point types in Java are unable to represent all possible values exactly.
- Equality comparison should be avoided!

- "Correct" comparison depends on the use case.
    - E.g. for small numbers you could check that the difference between two values is small enough

```
System.out.println(0.5f == 0.5); // Prints true
System.out.println(0.1f == 0.1); // Prints false
System.out.println((0.1f - 0.1) < 0.00000001); // Prints true
```

Boolean comparison

- In Java `boolean` is not a numeric type!
- There is no "order" on boolean values.
- They can’t be compared to other types.
- However, it is possible to check two booleans for (in-)equality using `==` and `!=`.

```
System.out.println(true ==  true); // prints "true"
System.out.println(true == false); // prints "false"
System.out.println(true !=  true); // prints "false"
System.out.println(true != false); // prints "true"
```

De Morgan’s Laws

- Sometimes we have to transform (complex) boolean formulas in order to…
    - … improve readability or
    - … increase efficiency

- Example: "De Morgan’s Laws"
    - A pair of rules that allow us to swap OR with AND (and vice-versa) using NOT.
    - Easy to prove using a truth table

Laws

```
!(A && B) == !A || !B
```

```
!(A || B) == !A && !B
```

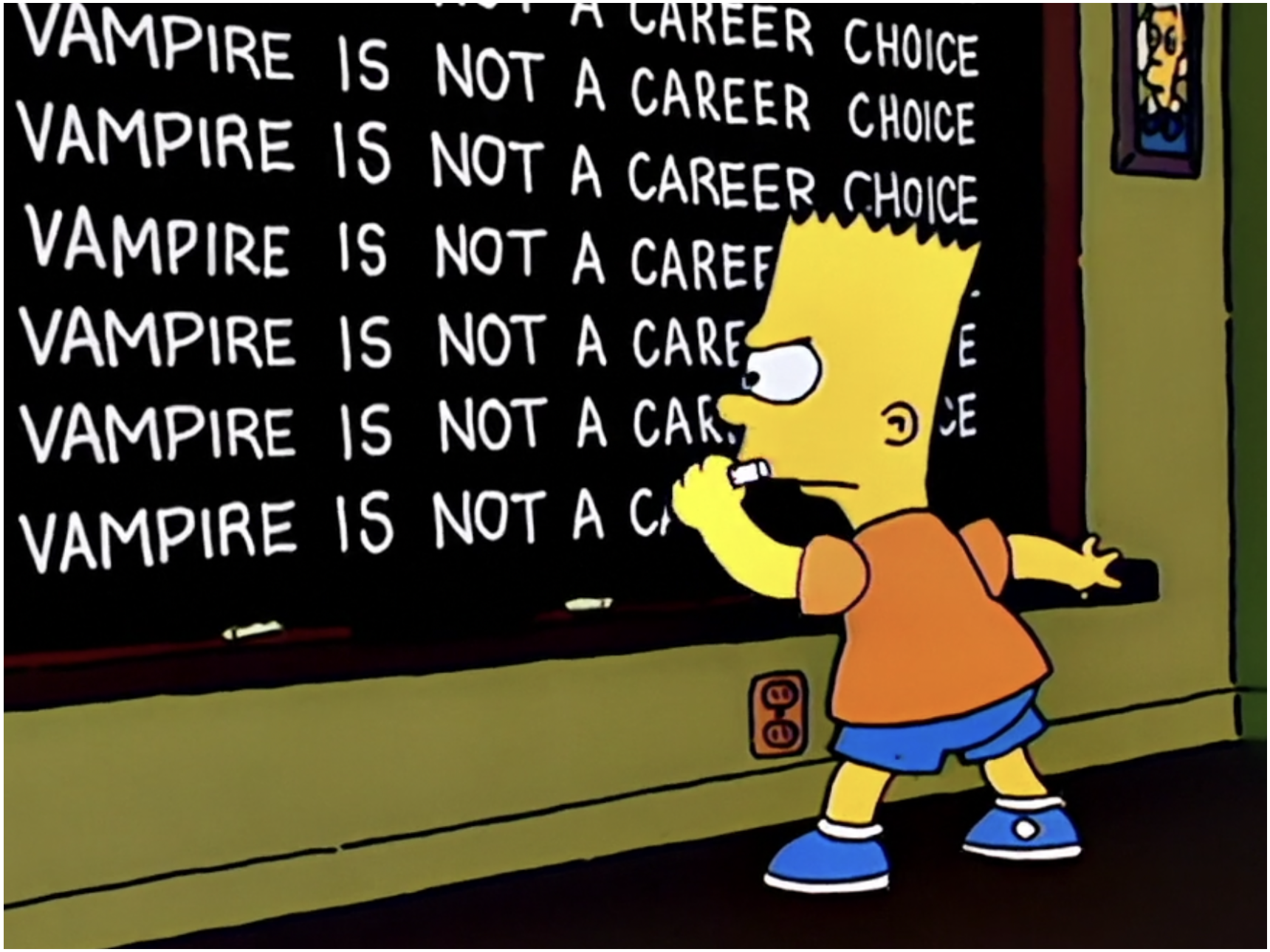
Truth Table

A	B	!A	!B	A && B	A || B	!(A && B)	!A || !B	!(A || B)	!A && !B
false	**false**	true	true	false	false	true	true	true	true
false	**true**	true	false	false	true	true	true	false	false
true	**false**	false	true	false	true	true	true	false	false
true	**true**	false	false	true	true	false	false	false	false

Module: Iteration

Iteration

Where computers do repetitive tasks. They have much more patience for this than we humans do.


VAMPIRE IS NOT A CAREER CHOICE
VAMPIRE IS NOT A CAREER CHOICE
VAMPIRE IS NOT A CAREER CHOICE
VAMPIRE IS NOT A CAREE
VAMPIRE IS NOT A CARE
VAMPIRE IS NOT A CAR
VAMPIRE IS NOT A C

Recap `if`

Remember `if`?

```
int value = 99;
if (value > 0) {
    System.out.println("Value " + value + " is bigger than 0!");
}
```

Output:

```
Value 99 is bigger than 0!
```

Introducing `while`

A `while`-loop is very similar to an `if`-statement:

```
int value = 99;
while (value > 0) {
    System.out.println("Value " + value + " is bigger than 0!");
}
```

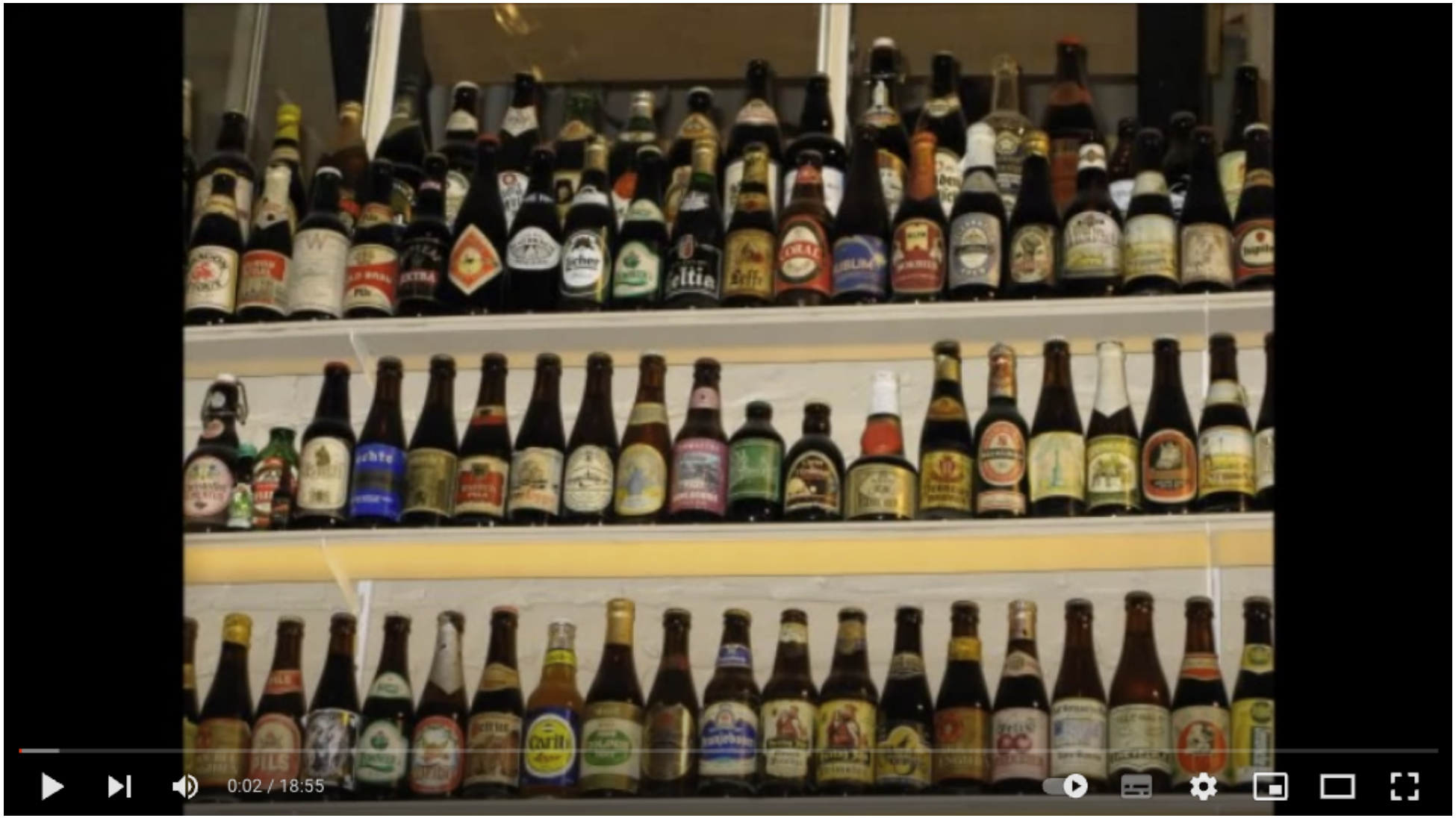
…but the output is very different:

```
Value 99 is bigger than 0!
Value 99 is bigger than 0!
Value 99 is bigger than 0!
...
```

```
Value 99 is bigger than 0!
Value 99 is bigger than 0!
Value 99 is bigger than 0!
```

Counting down

Computers also have the endless patience for things we humans can only tolerate on long drives.



https://www.youtube.com/watch?v=FITjBet3dio

Counting down

```
int bottlesOfBeer = 99;
while (bottlesOfBeer > 0) {
    System.out.println(bottlesOfBeer + " bottles of beer on the wall.");
    System.out.println(bottlesOfBeer + " bottles of beer.");

    bottlesOfBeer = bottlesOfBeer - 1;

    System.out.println("Take one down, pass it around, " + bottlesOfBeer + " bottles of beer on the wall.");
}
```

Output:

```
99 bottles of beer on the wall.
99 bottles of beer.
Take one down, pass it around, 98 bottles of beer on the wall.
...
2 bottles of beer on the wall.
2 bottles of beer.
Take one down, pass it around, 1 bottles of beer on the wall.
1 bottles of beer on the wall.
1 bottles of beer.
Take one down, pass it around, 0 bottles of beer on the wall.
```

Infinite loop

We can still make an infinite loop by accident. If we forget to count down, for example:

```
int bottlesOfBeer = 99;
```

```
while (bottlesOfBeer > 0) {
    System.out.println(bottlesOfBeer + " bottles of beer on the wall.");
    System.out.println(bottlesOfBeer + " bottles of beer.");

-   bottlesOfBeer = bottlesOfBeer - 1;

    System.out.println("Take one down, pass it around, " + bottlesOfBeer + " bottles of beer on the wall.");
}
```

No execution

We can also write a loop that doesn't execute at all. If we start counting at zero for example:

```
+int bottlesOfBeer = 0;
-int bottlesOfBeer = 99;
while (bottlesOfBeer > 0) {
    System.out.println(bottlesOfBeer + " bottles of beer on the wall.");
    System.out.println(bottlesOfBeer + " bottles of beer.");

    bottlesOfBeer = bottlesOfBeer - 1;

    System.out.println("Take one down, pass it around, " + bottlesOfBeer + " bottles of beer on the wall.");
}
```

Exercises

To summarize, we can use the `while` construct to keep repeating some piece of code until some condition goes from `true` to `false` (or don't execute it at all, if the condition was `false` to begin with).

Exercise Write a loop that prints numbers from 1000 to 0. How long does that take to execute? (Computers are pretty fast!)

Exercise Write a loop that prints numbers from 0 to 1000.

Exercise Write a loop that calculates the sum of the first 20 numbers (`1 + 2 + 3 + ... + 20`) and prints the result when it is done.

Guessing numbers

Guess my (secret) number!

```
// Pick a secret value:
int mySecretValue = 8;

// Make the user of the program guess:
int yourGuess = askUserForNumber();

// While the user has guessed wrong...
while (yourGuess != mySecretValue) {

    // Let him/her know:
    System.out.println("You guessed wrong!");

    // Ask for another guess:
    yourGuess = askUserForNumber();
}

System.out.println("You guessed right!");
```

where

```
// Don't worry about this for now:
Scanner fromUser = new Scanner(System.in);
int askUserForNumber() {
```

```
    return fromUser.nextInt();
}
```

(Don't worry! Almost everything about `askUserForNumber` will become clear by the end of the day.)

Guessing numbers with hints

We can give the user some hints, to practice our `if`-statements:

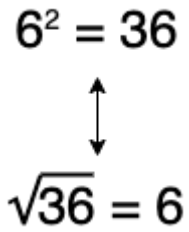
```
// Pick a secret value:
int mySecretValue = 8;

// Make the user of the program guess:
int yourGuess = askUserForNumber();

// While the user has guessed wrong...
while (yourGuess != mySecretValue) {

+   if (yourGuess < mySecretValue) {
+       System.out.println("Your guess was too low!");
+   }
+
+   if (yourGuess > mySecretValue) {
+       System.out.println("Your guess was too high!");
+   }
-   // Let him/her know:
-   System.out.println("You guessed wrong!");

    // Ask for another guess:
    yourGuess = askUserForNumber();
}
```

```
System.out.println("You guessed right!");
```

Square root

Using a similar guessing game, we can use `while` to calculate the square of a number, using only addition, multiplication and division.

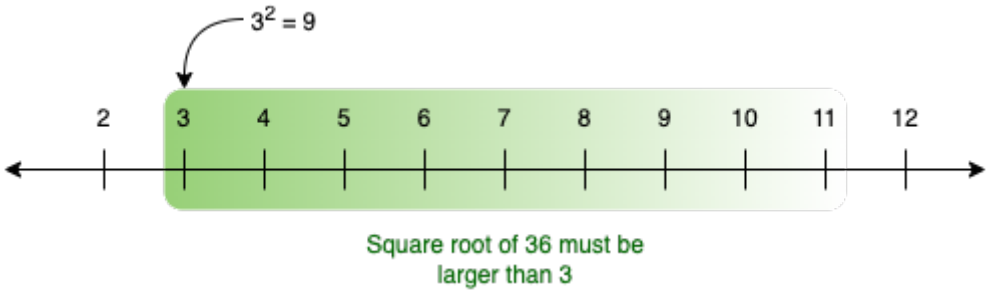
You probably know that:



But how to use this knowledge to actually calculate the square root from 36?

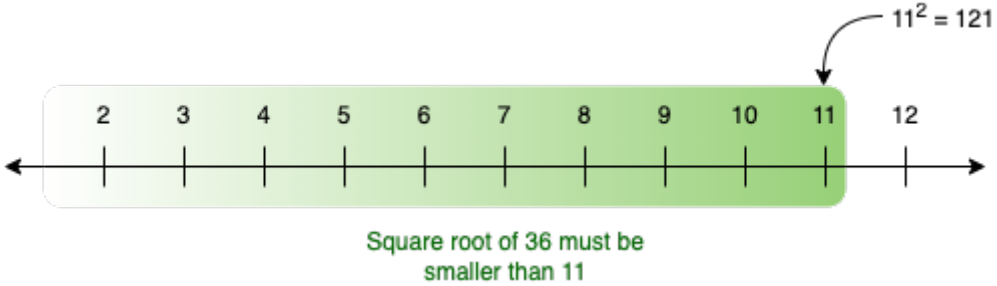
Wild guess (too low)

Let’s start by guessing the square root of 36 is 3. The guess is clearly wrong, because 3 * 3 = 9 which is much smaller than 36, but we did learn something by guessing: the square root of 36 must be some number that is *larger* than 3.



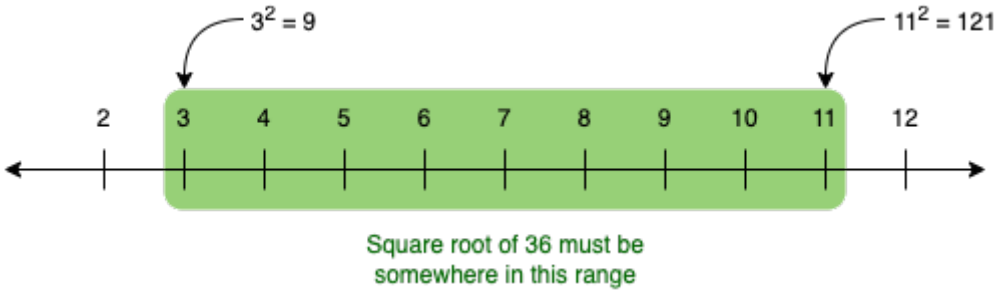
Wild guess (too high)

Let’s make another wild guess like 11, and check if it is the square root of 36. We immediately see that our guess is much too large, because 11 * 11 = 121 which is quite a bit larger than 36. The square root of 36 must be some number that is *smaller* than 11.



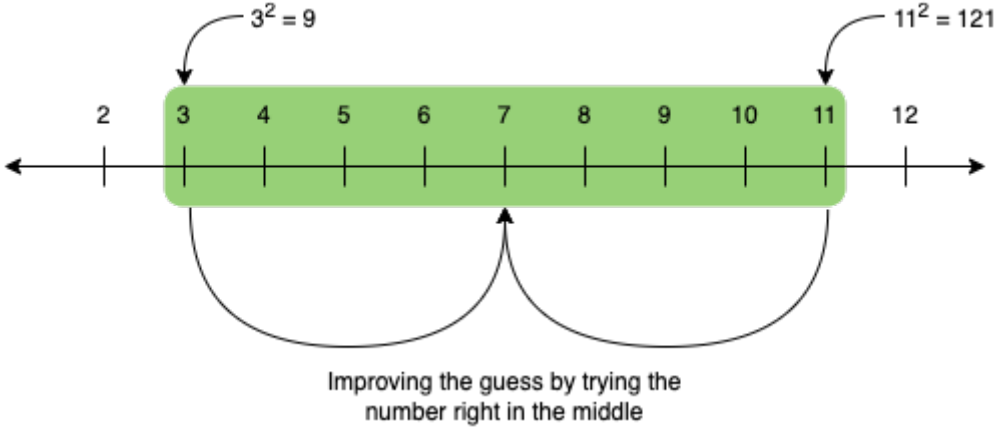
Guess range

Whatever the square root of 36 is, we know it must be larger than 3 but smaller than 11. The real square root of 36 must be somewhere in between those two numbers.



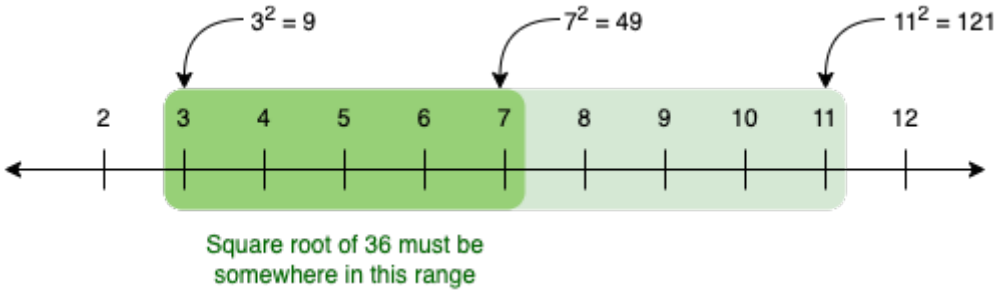
Improve upper bound

Right in the middle of the two lies the number 7 (calculated by taking the *average* of both numbers) which is probably closer to the square root than either 3 or 11.



Improve upper bound

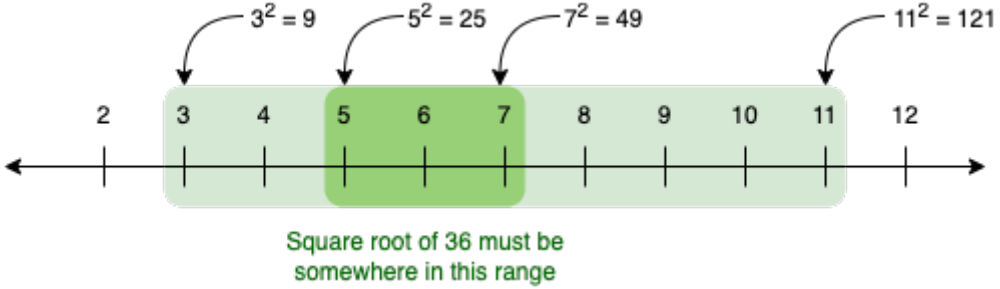
Not only must the square root of 36 be smaller than 11, we have learned that it is smaller than 7 too. This eliminates the numbers 11, 10, 9 and 8 from the range of possible square roots. The number 7 makes for a much better upper bound than 11.



Improve lower bound

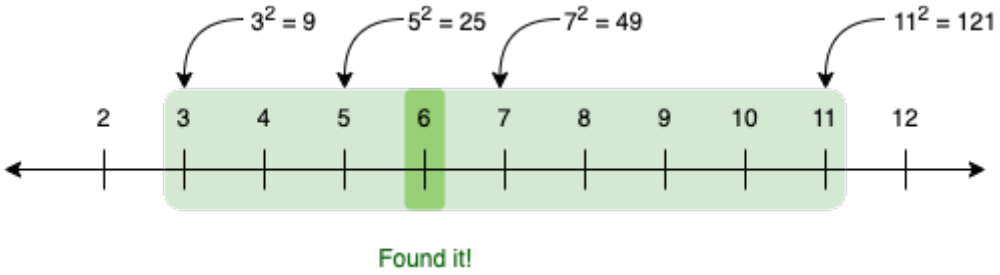
We can improve the guess even more, by checking the number right in the middle between 3 and 7 (average 5) and check if we are getting closer again.

This time, the improved guess is a bit too low. Although we still haven't foundt he square root, We did learn something in the process. We have learned that the square root of 36 must be larger than 5. This eliminates the number 3 and 4 from the range of possible square roots. The number 5 makes for a much better lower bound than 3.



One more time!

Let's repeat this process one more time. Right in the middle of 5 and 7 we find the number 6. Checking if that number is the square root of 36 reveals what we secretly already knew:



Square root in Java

Using a process that is very similar to a human guessing a (secret) value, we can let the computer guess at the square root of some number. Let's write a program to calculate the square root of 36:

```
double input = 36;

// The square root of any number is always bigger than zero:
double guessTooLow = 0;

// The quare root of any number is always smaller than that number itself:
double guessTooHigh = input;

// Our best guess is right in the middle of those two:
double bestGuess = (guessTooLow + guessTooHigh) / 2;

// Check if we are done:
boolean foundSquareRoot = bestGuess * bestGuess == input;

while (!foundSquareRoot) {

    // Print our best guess:
    System.out.println("Best guess: " + bestGuess);
```

```
    if (bestGuess * bestGuess < input) {
        guessTooLow = bestGuess;
    }

    if (bestGuess * bestGuess > input) {
        guessTooHigh = bestGuess;
    }

    // Improve our guess:
    bestGuess = (guessTooLow + guessTooHigh) / 2;

    // Check if we are done now:
    foundSquareRoot = bestGuess * bestGuess == input;
}
System.out.println("Found: " + bestGuess);
```

Square root in Java

This is a great example of an *algorithm*: a sequence of (repeatable) steps to calculate something interesting. Let's see what it prints:

```
Best guess: 18.0
Best guess: 9.0
Best guess: 4.5
Best guess: 6.75
Best guess: 5.625
Best guess: 6.1875
Best guess: 5.90625
Best guess: 6.046875
Best guess: 5.9765625
Best guess: 6.01171875
Best guess: 5.994140625
Best guess: 6.0029296875
Best guess: 5.99853515625
```

```
Best guess: 6.000732421875
...
Best guess: 6.0000000000001705
Best guess: 5.999999999999915
Best guess: 6.000000000000043
Best guess: 5.999999999999979
Best guess: 6.000000000000011
Best guess: 5.999999999999995
Best guess: 6.000000000000003
Best guess: 5.999999999999998
Found: 6.0
```

Exercises

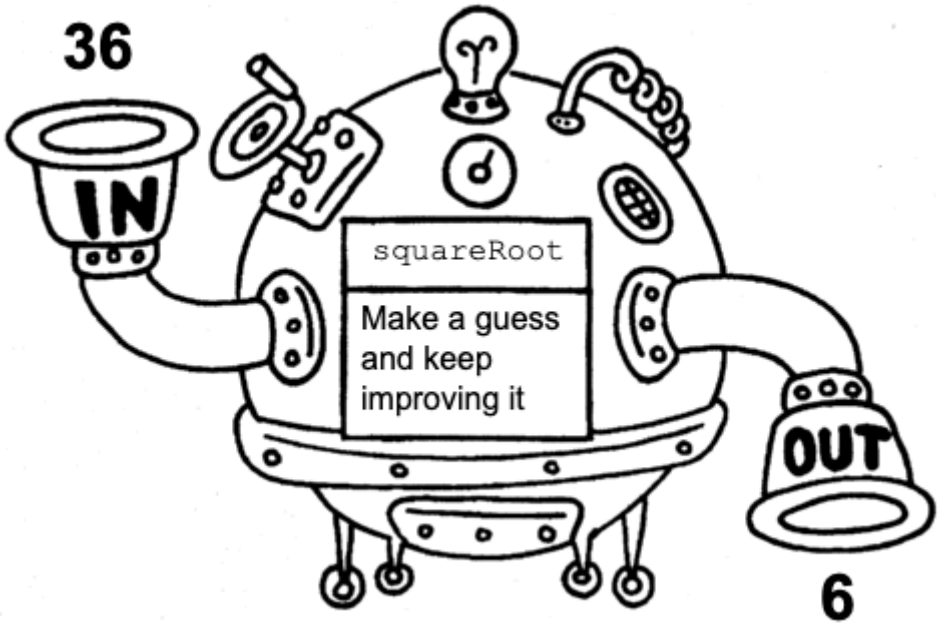
Exercise Use this algorithm to compute the square root of 49.

Exercise Use this algorithm to compute the square root of 121.

Module: functions

Functions

To avoid repeating this code over and over again we can package this algorithm as a *function*. A function is a way to assign a name to a re-usable set of instructions. In it's most general form, a function accepts some input and produces some output, much like the following little machine:



Square root as a function

The input is not some fixed value like `36` but is passed to the function as a *parameter* (a fancy word for the machine's input).

Instead of printing, the function ends with a `return` statement.

```
-double input = 36;
+double squareRoot(double input) {

     // The square root of a number is always bigger than zero:
     double guessTooLow = 0;

     // The quare root of a number is always smaller than that number itself:
     double guessTooHigh = input;
```

```
     // Our best guess is right in the middle of those two:
     double bestGuess = (guessTooLow + guessTooHigh) / 2;

     // Check if we are done:
     boolean foundSquareRoot = bestGuess * bestGuess == input;

     while (!foundSquareRoot) {

        // Print our best guess:
         System.out.println("Best guess: " + bestGuess);

         if (bestGuess * bestGuess < input) {
             guessTooLow = bestGuess;
         }

         if (bestGuess * bestGuess > input) {
             guessTooHigh = bestGuess;
         }

         // Improve our guess:
         bestGuess = (guessTooLow + guessTooHigh) / 2;

         // Check if we are done now:
         foundSquareRoot = bestGuess * bestGuess == input;
     }

-System.out.println("Found: " + bestGuess);
+    return bestGuess;
+}
```

Anatomy of functions (with a single input)

In general, a functions looks somewhat like this:

```
<return type> <name>(<parameter type> <parameter name>) {
    <body>
}
```

Calling/invoking functions

Now that we created this neat function to calculate the square root of any number we'd like to pass to it, let's see how we can *call* this function:

```
// Calculate the square root:
double output = squareRoot(49);

// Print:
System.out.println("Square root of 49 is " + output);
```

We call (or *invoke*) the function by typing its name, followed by the number we'd like to pass as input wrapped in parentheses.

Multiple `return` statements

A body can have multiple `return` statements. A `return` forces the function to return the value immediately and to skip execution any of the code (if any) after the `return` statement:

```
int ensurePositive(int possiblyNegative) {

    // If the input is negative, multiply by
```

```
    // -1 to make it positive and return.
    if (possiblyNegative < 0) {
        return -1 * possiblyNegative;
    }

    // If the input was negative, this wouldn't
    // be executed. Therefore, it is safe to
    // return the original input. It must be
    // positive (or zero).
    return possiblyNegative;
}
```

Multiple input values

You may have noticed that we repeat the expression `(guessTooLow + guessTooHigh) / 2` to calculate the "mid-point" of `guessTooLow` and `guessTooHigh` twice in the body of `squareRoot`.

Whenever you find yourself repeating an expression like that, you should pause and consider if it is worthwhile to extract that expression as a function:

```
double midPoint(double x, double y) {
    return (x + y) / 2;
}
```

Using `midPoint`

Let's update `squareRoot` to make use of this convenient function:

```
double squareRoot(double input) {

    // The square root of a number is always bigger than zero:
```

```
    double guessTooLow = 0;

    // The quare root of a number is always smaller than that number itself:
    double guessTooHigh = input;

    // Our best guess is right in the middle:
+   double bestGuess = midPoint(guessTooLow, guessTooHigh);
-   double bestGuess = (guessTooLow + guessTooHigh) / 2;

    // Check if we are done:
    boolean foundSquareRoot = bestGuess * bestGuess == input;

    while (!foundSquareRoot) {

        System.out.println("Best guess: " + bestGuess);

        if (bestGuess * bestGuess < input) {
            guessTooLow = bestGuess;
        }

        if (bestGuess * bestGuess > input) {
            guessTooHigh = bestGuess;
        }

        // Improve our guess:
+       bestGuess = midPoint(guessTooLow, guessTooHigh);
-       bestGuess = (guessTooLow + guessTooHigh) / 2;

        // Check if we are done now:
        foundSquareRoot = bestGuess * bestGuess == input;
    }

    return bestGuess;
}
```

No input values

Note that this little `midPoint`-machine accepts two values as input:

```
double midPoint(double x, double y) {
    return (x + y) / 2;
}
```

A function can accept any number of parameters. Even zero parameters is allowed (although the use of such a function is limited):

```
int returnSomeNumber() {
    return 42;
}
```

Why functions

Benefits:

- Re-use set of instructions
- Attach meaningful name to set of instructions
- Hide complexity behind friendly function name

Exercises

Exercise Create a function `max` that returns the bigger of two `double` parameters. Also write a similar function `min` that returns the smallest. Remember that it is OK to have multiple `return` statements in a function.

Exercise Write a function that calculates a person's body mass index. The body mass index (BMI) is defined as the body mass (kg) divided by the square of the body height (m). Choose the appropriate data type for the body mass and the height.

Exercise Write a function `factorial` that calculates the *factorial* of its input. The factorial of an `int` is the product of all positive integers less than or equal to that `int`. In other words:

```
factorial(5) == 1 * 2 * 3 * 4 * 5
```

Multiple output values

Although a function can accept multiple input values, a function can't return multiple values as output.

Even though we can't return multiple values, it is valid for a function to return no output at all. In that case, the return type is `void` (which indicates the *absence* of a return type):

```
void sayHello(String name) {
    System.out.println("Hello, " + name + "!");
}
```

Exercises

Exercise If your BMI is less than 18.5, you're considered underweight. If it's anywhere from 18.5 to 25 then you're considered normal. 25 to 30 is overweight and more than 30 is obese. Write a function `insult` that accepts a person's body mass and body height (like before) and prints the following insults based on the person's BMI:

BMI	Text
< 18.5	You’re underweight, you emo, you!
> 30.0	You’re a whale, congratulations!
otherwise	You’re supposedly normal. Pffft, I bet you’re ugly!

This function returns nothing.

Module: arrays

Running laps

Let’s say we have been running laps on the track all afternoon, and we have recorded how many seconds it took to complete one 400m lap:

```
int firstLap = 61;
int secondLap = 59;
int thirdLap = 67;
int fourthLap = 72;
int fifthLap = 68;
int sixthLap = 69;
```

Wow! It seems we are pretty fit!

Average lap time

You might want to calculate how long, on average, it took to complete a lap:

```
int averageTime = (firstLap + secondLap + thirdLap + fourthLap + fifthLap + sixthLap) / 6;
```

The more laps we run, the more annoying it will become to calculate the average lap time.

Imagine having to calculate the average of 100 laps or even 1000 laps this way!

Arrays

Of course, there is a better way!

Java offers a way to store multiple values in a single variable: arrays. An array is a data structure, which can store a *fixed number* of elements of the *same data type*.

```
int[] laps = {61, 59, 67, 72, 68, 69};
```

Not just `int`!

Arrays are not limited to `int` values. You can create an array from any simple data type. In a similar way, for example, we could store all the athletes on the track today:

```
String[] athletes = {"Alice", "Bob", "Carol", "Dave"};
```

Reading from arrays

You can read data from these arrays by using the *index* (the position in the array):

```
System.out.println("First athlete: " + athletes[1]);
```

The answer may not be what you expect! Instead of printing the first element of the array (which is `"Alice"`) it prints the second:

```
First athlete: Bob
```

Indices start at 0

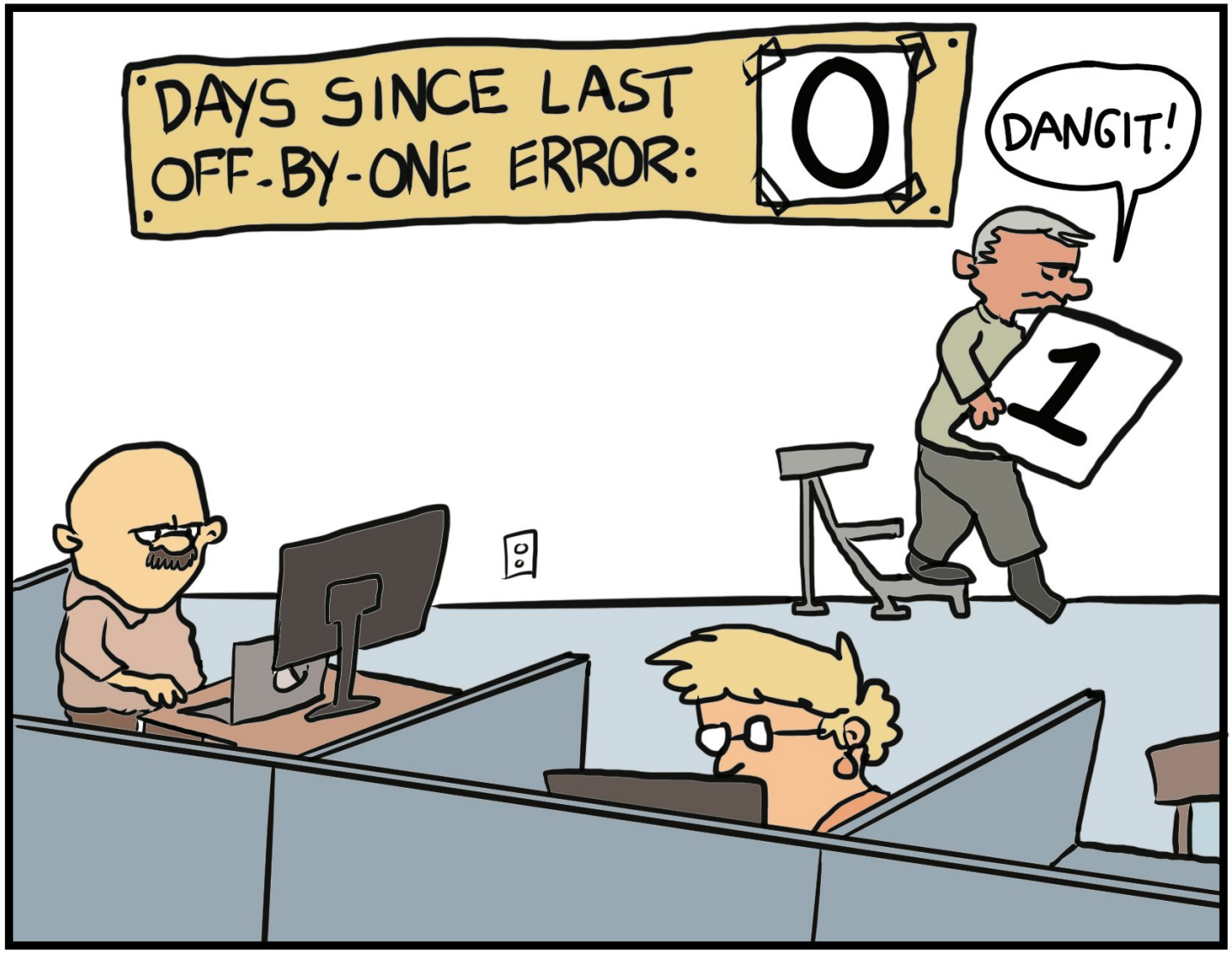
For historical reasons (that we don't want to go into right now) arrays in Java start with index (position) zero. The first element can be found at index 0:

```
System.out.println("First athlete: " + athletes[0]);
```

The valid indices of the `athletes` array are 0, 1, 2, and 3.

FaaS and Furious by Forrest Brazeal

A CLOUD GURU



Out of bounds

What will happen if we try to access the fifth element (at index 4)?

```
jshell> System.out.println("Fifth athlete: " + laps[4]);
|  Exception java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
|        at (#2:1)
```

Instead of printing the lap time, Java responds with a nasty error message.

Writing to arrays

Besides reading, we can use the index to change values in an array as well. The syntax is very similar to reading:

```
int[] phoneNumber = {0, 6, 2, 5, 5, 4, 4, 0, 2, 5};

// Change the third digit (index 2) of my phone number to 8:
phoneNumber[2] = 8;
```

We’ve all been there, right ;)

Reading and writing

We can even combine reading values and updating values in arrays:

```
// Increase the sixth digit (index 5) by one:
phoneNumber[5] = phoneNumber[5] + 1;

// Make second and fourth digit equal.
// Remember that indices are always off by one.
// This will take some time to get used to!
phoneNumber[1] = phoneNumber[3];
```

Creating array of known length

There is another way to create an array. Instead of providing the values directly, we can create an array with a fixed number of default values by using the `new` keyword:

```
// Create an array of six elements (all zeroes):
int[] defaultValues = new int[6];

// Same as above:
int[] defaultValues = {0, 0, 0, 0, 0, 0};
```

When we create an array this way, we need to specify the length of array between the square brackets. In the example above, we are creating an array of six integers.

Exercises

Exercise Print each value in an array of 3 default `String`s. The person who "invented" this default value calls it his "billion-dollar mistake", as he estimates that the confusion it creates has caused billions of dollars of damage of the years.

Why create an array this way?

An array that is filled with default values is not very useful on its own, but it will be quite convenient to be able to ``reserve" some space for a number of `int`s, `double`s or `String`s. Especially if the number of elements is large, and you don’t want to be stuck writing:

```
int[] longArray = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ..., 0}
```

We can assign the appropriate values (which can be the result of some complex calculation that its inconvenient to do on the spot) later.

Printing laps

Arrays and `while`-loops are a match made in heaven! By using a counter `index` that we increment each iteration, we can print the time it took to complete each lap:

```
// Define what the first- and last valid index is:
int firstIndex = 0;
int lastIndex = 5;

// Start at the first index:
int index = firstIndex;

// Keep looping if we have not reached the end of the array yet:
while (index <= lastIndex) {

    // Print item:
    System.out.println(laps[index]);

    // Move to the next lap:
    index = index + 1;
```

```
}
```

Total time on the track

With a similar loop, we can calculate the total time we spent on the track:

```
// Initialize total time to zero:
int totalTime = 0.0;

int index = firstIndex;
while (index <= lastIndex) {

    // Add time:
    totalTime = totalTime + laps[index];

    // Move to the next lap:
    index = index + 1;
}

System.out.println("Total time: " + totalTime);
```

Calculating the average

Because we ran six laps, the average time per lap is:

```
int averageTime = totalTime / 6;
```

That was precisely what we were looking for when we began talking about arrays! It's quite a bit of code, but the benefit is that it will work for arrays of hundreds and even thousands of items with little modification (we just need to change the value of `lastIndex` to match the array).

Exercises

Exercise Write a function `averageLapTime` that accepts an array of lap times (parameter of type `int[]`) and the number of laps we ran (parameter of type `int`) and returns the average time per lap (another `int`). Try it out with a long array.

Exercise Write a `while`-loop that identifies and prints the fastest lap (minimum value in `laps`). For `laps` it should return `59`.

Exercise Write a `while`-loop that identifies and prints the index of the fastest lap. For `laps` it should return `1`.

Exercise Write a `while`-loop that decrements each lap time by one second.

Exercise Create an array with 20 integers (initialized to their default values). Set the first two elements (index 0 and index 1) to 1. Using a `while` -loop, change all subsequent elements to contain the sum of the previous two elements. You may recognize this as the Fibonacci-sequence:

```
int[] fib = {1, 1, 2, 3, 5, 8, ..., 6765};
```

`for`-loop

There is a second type of loop, that makes it a bit more convenient to loop through an array. We are talking about the `for`-loop:

```
for (String athlete : athletes) {
    System.out.println("Athlete: " + athlete);
}
```

Similar to the following, more cumbersome `while`-loop:

```
int firstIndex = 0;
int lastIndex = 3;
```

```
int index = firstIndex;
while (index <= lastIndex) {
    String athlete = athletes[index];
    System.out.println("Athlete: " + athlete);
    index = index + 1;
}
```

Note: there are different kinds of `for`-loops, but we'll skip these for now.

Exercises

Although it is certainly more *convenient* than the `while`-loop (where we manually have to keep track of the `index` and make sure to increment it each iteration), it is also a bit more *limited*. By using a `for`-loop, we can't change the loop-value, and we don't have access to the current index.

Exercise Rewrite the function `averageLapTime` to use a `for`-loop. Do we still need the parameter that passes the length of the array to the function?

Exercise Write a `for`-loop that identifies and prints the fastest lap (minimum value in `laps`). For `laps` it should print `59`. Note that we tried this before (with a `while`-loop) in one of the previous exercises.

Exercise Is it possible to write a `for`-loop that identifies and prints the index of the fastest lap?

Exercise Is it possible to write a `for`-loop that decrements each lap time by one second?

Module: multiple arrays

Eggs

So far we've seen there are values of many types. We can use these to represent any interesting property of a purchase of two boxes of large eggs:

```
String product = "Large Eggs";
int quantity = 2;
double price = 0.99;
```

We can use these variables to determine what we have to pay for these:

```
double toPay = quantity * price;
```

Groceries

We seldom buy a single thing when we go to the supermarket. Usually we brought a whole shopping list of things to buy. Can we use Java to calculate how much we have to pay for all our groceries?

Many variables

Like before, it's inconvenient and unattractive to store each purchase in a single variable.

```
String product1 = "Large Eggs";
int quantity1 = 2;
double price1 = 0.99;
```

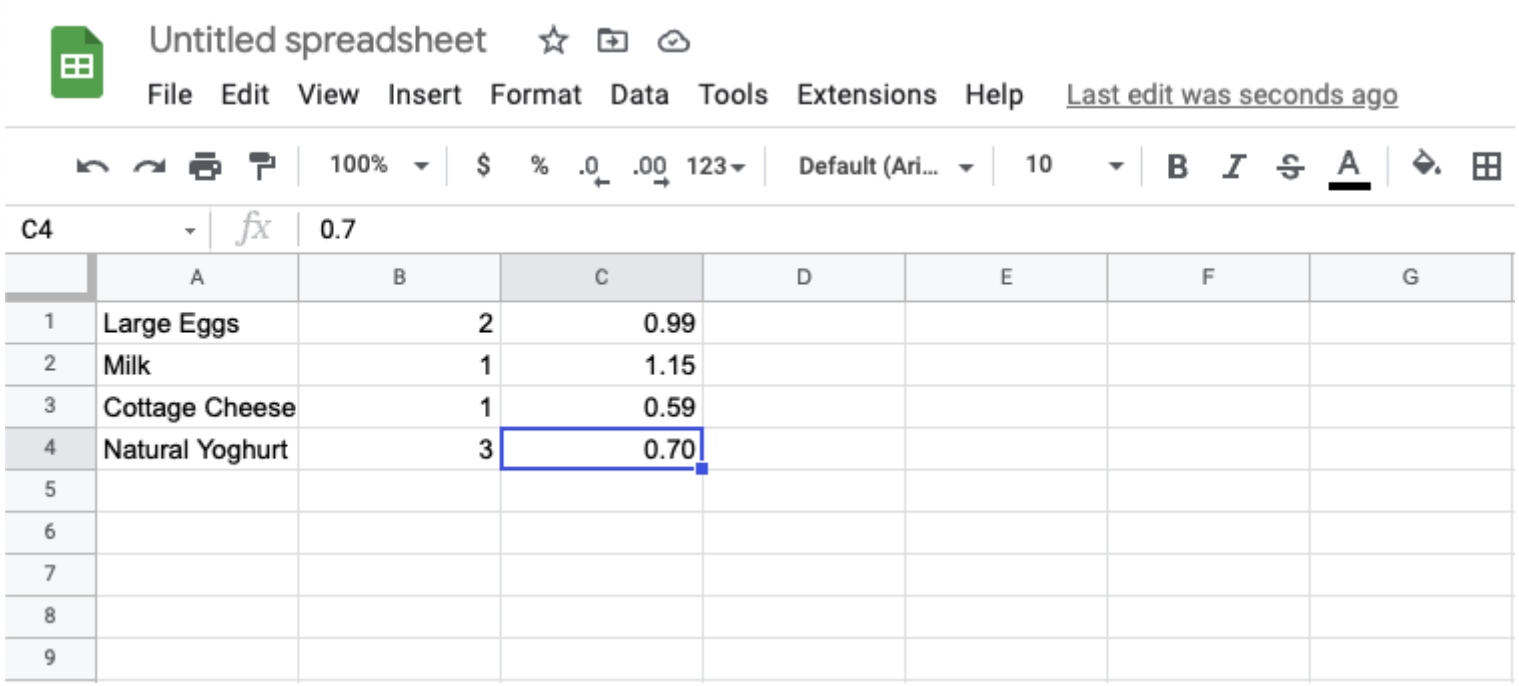
```
String product2 = "Milk";
int quantity2 = 1;
double price2 = 1.15;

String product3 = "Cottage Cheese";
int quantity3 = 1;
double price3 = 0.59;

String product4 = "Natural Yoghurt";
int quantity4 = 3;
double price4 = 0.70;

...

// To calculate the total price on the receipt:
double totalPrice = quantity1 * price1 + quantity2 * price2 + quantity3 * price3 + quantity4 * price4 + ....
```

Arrays

This is another great opportunity to apply what we have learned about arrays. We create an array for each quantity (limited to 4 items for convenience):

```
String[] products = {"Large Eggs", "Milk", "Cottage Cheese", "Natural Yoghurt"};
int[] quantities = {2, 1, 1, 3};
double[] prices = {0.99, 1.15, 0.59, 0.70};
```

This is similar to how you would store items in a spreadsheet. A variable like `products`, `quantities` or `discounts` corresponds to a "column". Values in the same "row" (same index) are related to the same purchase.



Print purchase

Using these four variables and a `while`-loop, we can print the values related to each of the purchases:

```
// Define what the first- and last valid index is:
int firstIndex = 0;
int lastIndex = 3;

// Start at the first index:
int index = firstIndex;

// Keep looping if we have not reached the end of the array yet:
while (index <= lastIndex) {
```

```
    // Print item:
    System.out.println("Product: " + products[index]);
    System.out.println("Count: " + quantities[index]);
    System.out.println("Price: " + prices[index]);

    // Move to the next item on the receipt:
    index = index + 1;
}
```

Calculating receipt's total price

With a similar loop, we can calculate the total price we have to pay:

```
// Initialize total price to zero:
double totalPrice = 0.0;

int index = firstIndex;
while (index <= lastIndex) {

    // Add price:
    totalPrice = totalPrice + quantities[index] * prices[index];

    // Move to the next item on the receipt:
    index = index + 1;
}

System.out.println("Total price: " + totalPrice);
```

That was precisely what we were looking for! Again, it's quite a bit of code, but like before the while loop works just as well for very long grocery lists.

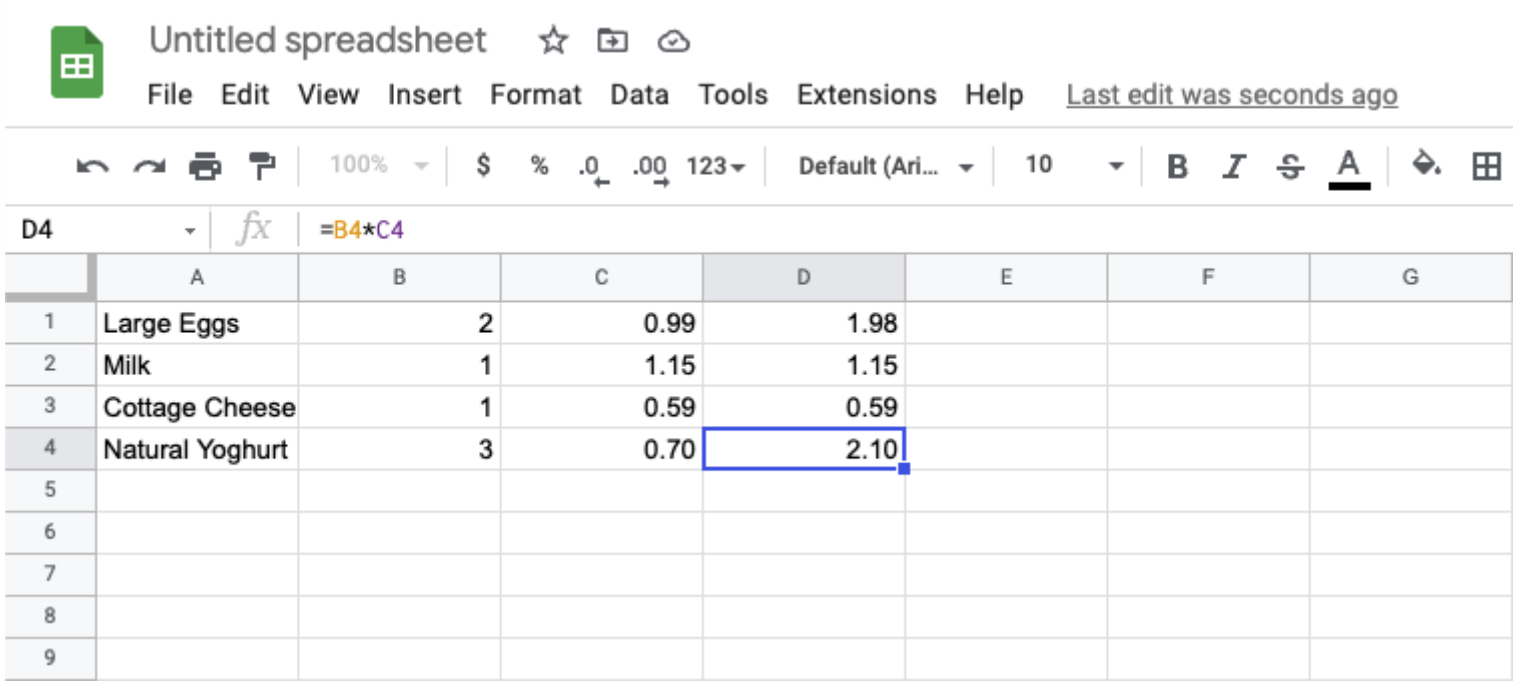
Sub-total

As a final example, we will show how to calculate the sub-total (quantity times price) for each item:

```
// Reserve some space (all elements are initialized to 0.0):
double[] subTotals = new double[4];

int index = firstIndex;
while (index <= lastIndex) {

    // Calculate sub-total:
    subTotals[index] = quantities[index] * prices[index];

    // Move to the next item on the receipt:
    index = index + 1;
}
```

Sub-total in spreadsheet

This is similar to adding an extra column in our spread sheet:



	A	B	C	D	E	F	G
1	Large Eggs	2	0.99	1.98			
2	Milk	1	1.15	1.15			
3	Cottage Cheese	1	0.59	0.59			
4	Natural Yoghurt	3	0.70	2.10			
5							
6							
7							
8							
9							

Note how, in the spreadsheet, the values in column D are the product of the values in column B (quantity) and column C (price).

Exercises

Exercise Write a `while`-loop that that identifies the product with the highest sub-total (quantity times price). Print the name of that product.

Exercise Is it possible to write a `for`-loop that identifies the product with the highest sub-total (quantity times price)? And if you can't use the `subTotals` array?

Exercise Using the appropriate loop, create an array with indices for `products`, `quantities` and `prices` (the result should be equal to `{0, 1, 2, 3}`)

Exercise Using the array with indices from the previous exercise, write a `for`-loop that identifies the product with the highest sub-total (quantity times price).

Module: Objects

Downsides of storing product, quantity and price in different variables

Having the data for each purchase stored in three different arrays is a bit cumbersome. We can't just use a four loop to determine the purchase with the highest sub-total, for example. We either have to calculate the sub-total for each purchase and store the result in a separate array or create an array of indices (both of which involve a `while`-loop).

Although the name of the product, the quantity and the price all belong to the same purchase, they are stored in three different array variables. They are only loosely associated (the product, quantity and price for the same purchase is located at the same index). Because a `for` loop doesn't provide the index (just the current item), it wasn't always possible to use such a loop to calculate what we need.

Introducing `class`

We can group data together using a `class`:

```
class Purchase {
    String product;
    int quantity;
    double price;
}
```

This defines a new data type `Purchase`. From now on, we'll be able to define variables of type `Purchase` (we will learn how to initialize those in a bit):

```
// Purchase is a new data type:
Purchase purchase = ...;
```

Creating a `Purchase`

`Purchase` is a bit more sophisticated than the data types like `int`, `double` and `boolean` that we have seen so far. A single value of data type `Purchase` provides access to three so-called "properties", which are `product` (type `String`), `quantity` (type `int`) and `price` (type `double`). We use the `new` keyword to create a new `Purchase` value, and we use the period (`.`) on it to access the purchase's properties:

```
// Create a new value of type `Purchase`, which contains three properties:
Purchase largeEggs = new Purchase();

// We can access the properties with using the "period operator":

// Will be 0 initially (default value):
System.out.println(largeEggs.quantity);

largeEggs.product = "Large Eggs";
largeEggs.quantity = 2;
largeEggs.price = 0.99;

// Will be 2 after assignment above:
System.out.println(largeEggs.quantity);
```

As you can see, by creating a `Purchase` with `new`, the fields are initialized to the same default values we recognize from arrays.

Exercises

Exercise Create a variable of type `Purchase[]` (a `Purchase`-array) that is equivalent to the three variables `products`, `quantities` and

`prices` above.

Exercise Use this `Purchase[]` and a `for`-loop to identify the purchase with the highest sub-total (quantity times price). Which is easier to use, a single array with a composite data type (`Purchase[] purchases`) or three separate arrays (`String[] products`, `int[] quantities` and `double[] prices`)? Which is easier to initialize?

Objects and primitives

Objects

- `Purchase`
- `String`
- Any type of array

Primitives

- `int`
- `double`
- `boolean`
- `char`

Constructing a `String` with `new`

We can create a `String` with the `new` keyword too:

```
String str = new String();
```

This new `String` will be empty:

```
// Same as above:
String str = "";
```

Constructing an array with new

The fact that we used the `new` keyword to initialize an array earlier hinted at the fact that arrays are objects already:

```
int[] allZeroes = new int[4];
```

Properties on `String` and array

Does that mean that a `String` and an array have properties too? Certainly! We could ask an array for its length, for example:

```
int numberOfProducts = products.length;
```

Sadly, we can't re-assign the `length` to make the array grow or shrink:

```
jshell> products.length = 5;
|  Error:
|  cannot assign a value to final variable length
|  products.length = 5;
|  ^-------------^
```

Exercises

Exercise Rewrite the function `averageLapTime` we wrote a long time (using a `while`-loop) to use the `length` property on the input array. Do we still need the second input for the array's length?

A note about `null`

The default value for objects is a special value `null` which indicates the *absence* of an object. When we try to access a property on such a non-object we get a nasty error message:

```
jshell> Purchase purchase = null;
purchase ==> null

jshell> purchase.quantity
|  Exception java.lang.NullPointerException: Cannot read field "quantity" because "REPL.$JShell$15B.purchase"
is null
|        at (#5:1)
```

A note about `null`

Because both `String` and `Purchase` are objects, and the default value for objects is `null`, an unitinialized `String[]` or `Purchase[]` will contain only `null`-values:

```
String[] products = new String[3];
for (String product : products) {
    System.out.println("Product: " + product);
}
```

Will print:

```
Product: null
Product: null
Product: null
```

You may remember this from the section on arrays.

Exercises

Exercise Write a function `findProduct` that accepts an array of purchases (`Purchase[]`) and the name of a product to find (`String`) and returns the `Purchase` with that name. What should we return if there is no `Purchase` with the given name?

Exercise Create a `class` called `Receipt` that holds the data for the entire receipt (a property `purchases` of type `Purchase[]`, and a property `totalPrice` of type `double`). Make sure that the `totalPrice` is the sum of the sub-total of each purchase. As you can see you can build more complex objects out of simple objects.

Defining methods

A `class` can contain values of type `int`, `double` and `String` (called properties) but it can also contain functions (called methods).

```
class Purchase {
    String product;
    int quantity;
    double price;

    String shout(String input) {
        return input + "!";
    }
```

```
}
```

You can call these functions/methods using the period-operator too:

```
Purchase purchase = new Purchase();
System.out.println(purchase.shout("Hello"));
```

About `System.out.println(…)`

Now that you have learned about methods, it is time to examine what we've been doing with `System.out.println(…)`. We are now able to recognize that `println` is actually a method on some `out` object, which is itself a property of the `System` object (this is not 100% accurate, but close enough).

Access to class variables

In addition to the parameters (like `input`), this method has access to the properties of the object it is called on too (like `product`, `quantity` and `price`):

```
class Purchase {
    String product;
    int quantity;
    double price;

    String shout(String input) {
        return input + "!";
    }

    double getSubTotal() {
        return quantity * price;
```

```
    }
}
```

Access to `this`

A method also has access to the object it is called on itself with the `this`-variable. The `this` variable is not declared anywhere (by you) but is provided by Java automatically.

```
class Purchase {
    String product;
    int quantity;
    double price;

    String shout(String input) {
        return input + "!";
    }

    double getSubTotal() {
-       return quantity * price;
+       return this.quantity * this.price;
    }
}
```

Some people think this is clearer, because you can always see where `quantity` and `price` come from (they are properties on `this` instead of some variables you defined earlier).

Updating properties

A method can even be used to update properties:

```
class Purchase {
    String product;
    int quantity;
    double price;

    String shout(String input) {
        return input + "!";
    }

    double getSubTotal() {
        return this.quantity * this.price;
    }

    void increaseQuantity() {
        this.quantity = this.quantity + 1;
    }
}
```

Updating properties

```
Purchase purchase new Purchase();
purchase.product = "Large Eggs";
purchase.quantity = 2;
purchase.price = 0.99;

// Prints "1.98"
System.out.println(purchase.getSubTotal());

purchase.increaseQuantity();

// Now prints "2.97"
System.out.println(purchase.getSubTotal());
```

Why methods

Benefits:

- Grouping methods that act on a common set of variables (like `product`, `quantity`, `price`) together.
- "Encapsulation" (we will talk about this later)

Exercises

Exercise Move the function `findProduct` into the class `Receipt`. Instead of finding a `Purchase` from an array we pass as input, it should look for it in the property `purchases` defined in `Receipt`.

Exercise Replace the property `totalPrice` with a method `getTotalPrice` that calculates the total price of the receipt (using each `Purchase's` `getSubTotal` method) when requested.

Methods on `String`

As proper objects, both `String` and the array have methods too:

```
String sugarPlease = "Can you pass me the sugar?"
char secondChar = sugarPlease.charAt(1);
```

The `charAt` method on `String` returns the character that is found at the index you provide (index `1` means the second character, remember!?)

Methods on `String`

Another interesting method on `String` is `toUpperCase`:

```
String sugarPlease = "Can you pass me the sugar?"
String giveMeSugar = sugarPlease.toUpperCase();
```

It will return a new `String` with all the caracters converted to upper-case. As you can see it is pretty convenient if methods like `toUpperCase` have access to the characters in the `String`. These methods can muck around with low-level character manipulation so you don't have to!

Constructors

There is a special kind of method that we can add to a class, which is called a *constructor*. A constructor is a method that is called whenever a new object is constructed. This special method has *no* return type (not even `void`) and must have the exact same name as the class itself (including the capital letter):

```
class Purchase {
    String product;
    int quantity;
    double price;

    // Note the name and the lack of return type:
    Purchase() {
        System.out.println("New purchase constructed!");
    }

    String shout(String input) {
        return input + "!";
    }

    double getSubTotal() {
        return this.quantity * this.price;
    }

    void increaseQuantity() {
```

```
        this.quantity = this.quantity + 1;
    }
}
```

Using constructors

Whenever we create a purchase, we see something printed to the screen:

```
jshell> Purchase purchase = new Purchase();
New purchase constructed!
purchase ==> Purchase@1a93a7ca
```

Why use constructors?

A constructor is a great place to initialize the object to some default state (which might be a little bit different from the values that each property is assigned by default). For example, we may prefer to initialize the `product` to an empty `String` or some other placeholder instead of `null`:

```
class Purchase {
    String product;
    int quantity;
    double price;

    Purchase() {
        System.out.println("New purchase constructed!");
        product = "---";
        quantity = 0;
        price = 0.0;
    }

    // Rest of the class like before...
```

```
}

Purchase purchase = new Purchase();
System.out.println("Product: " + purchase.product);
System.out.println("Quantity: " + purchase.quantity);
System.out.println("Price: " + purchase.price);
```

Passing input to constructor as parameters

Just like a regular method, a constructor can accept some parameters to make initializing easier:

```
class Purchase {
    String product;
    int quantity;
    double price;

    Purchase(String initialProduct, int initialQuantity, double initialPrice) {
        System.out.println("New purchase constructed!");
        product = initialProduct;
        quantity = initialQuantity;
        price = initialPrice;
    }

    // Rest of the class like before...
}
```

These three input values need to be provided when you create a `Purchase` with `new`.

Calling constructors with parameters

A constructor makes constructing a `Purchase` much easier. We just need to provide an initial product, quantity and price when constructing the object (much like calling a function):

```
Purchase[] purchases = [
    new Purchase("Large Eggs", 2, 0.99),
    new Purchase("Milk", 1, 1.15),
    new Purchase("Cottage Cheese", 1, 0.59),
    new Purchase("Natural Yoghurt", 3, 0.70)
];
```

This is much more convenient than the equivalent:

```
Purchase purchases = new Purchase[4];
purchases[0] = new Purchase();
purchases[0].product = "Large Eggs";
purchases[0].quantity = 2;
purchases[0].price = 0.99;

...
```

Exercises

Exercise Add a constructor to the class `Receipt` from before. It should accept an array of `Purchase`-objects, but nothing else.

Exercise The constructor of a class is a nice place to check if the `Purchase` that we're about to create is in a valid state. In the constructor, ensure that both the `quantity` and the `price` are larger than zero. Print a helpful error message otherwise.

Multiple constructors

A class can have multiple constructors. This is useful if there are multiple ways of initializing an instance of that class:

```
class Purchase {
    String product;
    int quantity;
    double price;

    Purchase() {
        System.out.println("New empty purchase constructed!");
    }

    Purchase(String initialProduct, int initialQuantity, double initialPrice) {
        System.out.println("New purchase constructed!");
        product = initialProduct;
        quantity = initialQuantity;
        price = initialPrice;
    }

    // Rest of the class like before...
}
```

Second constructor for `String`

Like `Purchase`, a `String` has several different constructors. We've seen one already (without any parameters, initializing to an empty string) but there are a number of other, more useful constructors. One, in particular, reveals how `String` is basically a (thin) wrapper class around an array of characters:

```
char data[] = {'a', 'b', 'c'};
```

```
String abc = new String(data);

// Same as above:
String abc = "abc";
```

If we want to, we can get those characters out again by using the `toCharArray` method on `String`:

```
// Equal to {'a', 'b', 'c'};
char[] charsFromAbc = abc.toCharArray();
```

Exercises

Exercise Create a `String`, extract its characters using `toCharArray`, reverse that array and use it to construct a new (reversed) `String`. This is similar to what the `reverse`-method does on `String`.