# **Codeacademy**

# **Les 1**

# **Java Review: Putting It All Together**

In this lesson, we’ve started writing our first programs in Java.

We’ve also learned rules and guidelines for how to write Java programs:

* Java programs have at least one class and one main() method.
  + Each class represents one real-world idea.
  + The main() method runs the tasks of the program.
* Java comments add helpful context to human readers.
* Java has whitespace, curly braces, and semicolons.
  + Whitespace is for humans to read code easily.
  + Curly braces mark the scope of a class and method.
  + Semicolons mark the end of a statement.
* Java is a compiled language.
  + Compiling catches mistakes in our code.
  + Compilers transform code into an executable class.

**Les 2**

The int data type allows values between -2,147,483,648 and 2,147,483,647, inclusive.

The double primitive data type can help. double can hold decimals as well as very large and very small numbers. The maximum value is 1.797,693,134,862,315,7 E+308, which is approximately 17 followed by 307 zeros. The minimum value is 4.9 E-324, which is 324 decimal places!

The char data type can hold any character, like a letter, space, or punctuation mark.

Char = single quotes in declaratie variabele

String = double quotes in declaratie variabele

# **Review**

Creating and filling variables is a powerful concept that allows us to keep track of all kinds of data in our program.

In this lesson, we learned how to create and print several different data types in Java, which you’ll use as you create bigger and more complex programs.

We covered:

* int, which stores whole numbers.
* double, which stores bigger whole numbers and decimal numbers.
* boolean, which stores true and false.
* char, which stores single characters using single quotes.
* String, which stores multiple characters using double quotes.
* *Static typing*, which is one of the safety features of Java.
* Variable naming conventions.

Practice declaring variables and assigning values to make sure you have a solid foundation for learning more complicated and exciting Java concepts!

**Les 3**

# **Review**

What’s the use of having variables if you can’t do anything with them? We’ve now seen some ways you can operate on variables and compare them. The possibilities are endless!

We covered:

* Addition and subtraction, using + and -
* Multiplication and division, using \* and /
* The modulo operator for finding remainders, %
* Greater than, >, and less than, <
* Equal to, ==, and not equal to, !=
* Greater than or equal to, >=, and less than or equal to, <=
* equals() for comparing Strings and other objects
* Using + to concatenate Strings

Practice some of these concepts here, to make sure you have a solid foundation for learning more complicated and exciting Java concepts!

**Les 4 Classes**

A *class* is the set of instructions that describe how an instance can behave and what information it contains.

Nieuwe class:

Classname (=type) variable name (=name) new Classname()

# **Classes: Instance Fields**

Our last exercise ended with printing an instance of Store, which looked something like Store@6bc7c054. The first part, Store, refers to the class, and the second part @6bc7c054 refers to the instance’s location in the computer’s memory.

We don’t care about memory location, but our instances have no other characteristics!

We’ll add associated data to an object by introducing *instance variables*, or *instance fields*. Instance fields are the state in our objects.

# **Classes: Constructor Parameters**

We’ll use a combination of constructor method and instance field to create instances with individual state.

We need to alter the constructor method because now it needs to access data we’re assigning to the instance.

Our Car constructor now has a *parameter*: String carColor.

public class Car { String color; // constructor method with a parameter public Car(String carColor) { // parameter value assigned to the field color = carColor; } public static void main(String[] args) { // program tasks }}

We need a value for the instance field color, so we’ve added String carColor as a parameter.

Parameters specify the type and name of data available for reference within a method’s scope.

We’ve already seen a parameter in the main() method: String[] args, but this is the first time we’re using the parameter value within a method body.

The parameter carColor references the value passed in during a method call:

new Car("blue");// carColor references "blue" inside constructornew Car("yellow");// carColor references "yellow" inside constructor

Within the constructor, we assign the parameter value to the instance field.

Instance fields are available for assignment inside the constructor because we declared them within the class.

STATE = variabele die je in de class zelf declareert. Die kun je gebruiken om instances met een eigen state te maken. Deze state krijgt de nieuwe class via de aanroep van de constructor in de main method. De state bestaat in feite uit de properties/value-paren die een instance heeft gekregen.

Classname nieuweclassnaam = new Classname(aanroep met waarde per parameter)

**Behaviour = methods**

**JAVA: INTRODUCTION TO CLASSES**

# **Classes: Review**

Java is an object-oriented programming language where every program has at least one class. Programs are often built from many classes and objects, which are the instances of a class.

Classes define the state and behavior of their instances. Behavior comes from methods defined in the class. State comes from instance fields declared inside the class.

Classes are modeled on the real-world things we want to represent in our program. Later we will explore how a program can be made from multiple classes. For now, our programs are a single class.

public class Dog { // instance field String breed; // constructor method public Dog(String dogBreed) { /\* value of parameter dogBreed assigned to instance field breed \*/ breed = dogBreed; } public static void main(String[] args) { /\* create instance: use 'new' operator and invoke constructor \*/ Dog fido = new Dog("poodle"); /\* fields are accessed using: the instance name, `.` operator, and the field name. \*/ fido.breed; // "poodle" }}

**Instructions**

The text editor contains a Dog class. Play around with the code!

Try to add and remove instance fields. Create instances with different values. Access and print different fields.

**Les 5 Methods**

Void = als een method niets retourneert als deze wordt aangeroepen!

Je moet wel aangeven wat een method retourneert,

public int numberOfTires() { int tires = 4; return tires; }

Met de ToString method

public String toString(){

return "This store sells " + productType + " at a price of " + price +".";

}

Kun je de waarden van een object uitprinten

Gebruik je hem niet, dan krijg je alleen de instances te zien, zoals ze zijn opgeslagen in de vorm van een soort ID:

Store@2aae9190Store@2f333739

# **Review**

Great work! Methods are a powerful way to abstract tasks away and make them repeatable. They allow us to define behavior for classes, so that the Objects we create can do the things we expect them to. Let’s review everything we have learned about methods so far.

* *Defining a method* : Methods have a method signature that declares their return type, name, and parameters
* *Calling a method* : Methods are invoked with a . and ()
* *Parameters* : Inputs to the method and their types are declared in parentheses in the method signature
* *Changing Instance Fields* : Methods can be used to change the value of an instance field
* *Scope* : Variables only exist within the domain that they are created in
* *Return* : The type of the variables that are output are declared in the method signature

As you move through more Java material, it will be helpful to frame the tasks you create in terms of methods. This will help you think about what inputs you might need and what output you expect.

Return is niet per se nodig om een variabele te kunnen benaderen. Je kunt hem in de global space declareren en dan aanpassen met een method. In de main method kun je dan de method aanroepen die de variabele aanpast en deze vervolgens printen met dotnotatie in de instance.

**Les 6 control flow**

**The first** condition to evaluate to true will have that code block run. Here’s an example demonstrating the order:

int testScore = 72;if (testScore >= 90) { System.out.println("A");} else if (testScore >= 80) { System.out.println("B");} else if (testScore >= 70) { System.out.println("C");} else if (testScore >= 60) { System.out.println("D");} else { System.out.println("F");}// prints: C

This chained conditional statement has two conditions that evaluate true. Because testScore >= 70 comes before testScore >= 60, only the earlier code block is run.

**Note:** Only one of the code blocks will run.

# **Review**

Before this lesson, our code executed from top to bottom, line by line.

Conditional statements add branching paths to our programs. We use conditionals to make decisions in the program so that different inputs will produce different results.

Conditionals have the general structure:

if (condition) { // consequent path} else { // alternative path}

Specific conditional statements have the following behavior:

* if-then:
  + code block runs if condition is true
* if-then-else:
  + one block runs if conditions is true
  + another block runs if condition is false
* if-then-else chained:
  + same as if-then but an arbitrary number of conditions
* switch:
  + switch block runs if condition value matches case value

**ins**

# Les 7 **Introduction to Conditional Operators**

**CONDITIONAL OPERATORS**

# **Review**

Conditional operators work on boolean values to simplify our code. They’re often combined with conditional statements to consolidate the branching logic.

Conditional-AND, &&, evaluates to true if the booleans on *both sides* are true.

if (true && false) { System.out.println("You won't see me print!");} else if (true && true) { System.out.println("You will see me print!");}

Conditional-OR, ||, evaluates to true if one or both of the booleans on either side is true.

if (false || false) { System.out.println("You won't see me print!");} else if (false || true) { System.out.println("You will see me print!");}

Logical-NOT, !, evaluates to the opposite boolean value to which it is applied.

if (!false) { System.out.println("You will see me print!");}

**Les** at **8 LEARN JAVA: ARRAYS**

Array heeft { } in Java. Krijgt ter predefinitie []

Bij method geen = ter declaratie

Java print een array niet zomaar uit: [Ljava.lang.String;@2aae9190s

# **Importing Arrays**

When we printed out the array we created in the last exercise, we saw a memory address that did not help us understand what was contained in the array.

If we want to have a more descriptive printout of the array itself, we need a toString() method that is provided by the Arrays *package* in Java.

import java.util.Arrays;

We put this at the top of the file, before we even define the class!

When we import a package in Java, we are making all of the methods of that package available in our code.

The Arrays package has many useful methods, including Arrays.toString(). When we pass an array into Arrays.toString(), we can see the contents of the array printed out:

# **Get Element By Index**

Now that we have an array declared and initialized, we want to be able to get values out of it.

We use square brackets, [ and ], to access data at a certain index:

double[] prices = {13.1, 15.87, 14.22, 16.66};System.out.println(prices[1]);

This command would print:

15.87

Because 15.87 is the item at the 1 index of the array. Remember that arrays start at index 0!

# **Creating an Empty Array**

We can also create empty arrays and then fill the items one by one. Empty arrays have to be initialized with a fixed size:

String[] menuItems = new String[5];

Once you declare this size, it cannot be changed! This array will always be of size 5.

After declaring and initializing, we can set each index of the array to be a different item:

menuItems[0] = "Veggie hot dog";menuItems[1] = "Potato salad";menuItems[2] = "Cornbread";menuItems[3] = "Roasted broccoli";menuItems[4] = "Coffee ice cream";

This group of commands has the same effect as assigning the entire array at once:

String[] menuItems = {"Veggie hot dog", "Potato salad", "Cornbread", "Roasted broccoli", "Coffee ice cream"};

We can also change an item after it has been assigned! Let’s say this restaurant is changing its broccoli dish to a cauliflower one:

menuItems[3] = "Baked cauliflower";

Now, the array looks like:

["Veggie hot dog", "Potato salad", "Cornbread", "Baked cauliflower", "Coffee ice cream"]

The args parameter is another example of a String array. In this case, the array args contains the arguments that we pass in from the terminal when we run the class file. (So far, args has been empty.)

So how can you pass arguments to main()? Let’s say we have this class HelloYou:

public class HelloYou { public static void main(String[] args) { System.out.println("Hello " + args[0]); }}

When we run the file HelloYou in the terminal with an argument of "Laura":

java HelloYou Laura

We get the output:

Hello Laura

The String[] args would be interpreted as an array with one element, "Laura".

When we use args[0] in the main method, we can access that element like we did in HelloYou.

Args is de verbinding tussen terminal input en java output

# **Review**

We have now seen how to store a list of values in arrays. We can use this knowledge to make organized programs with more complex variables.

Throughout the lesson, we have learned about:

* Creating arrays explicitly, using { and }.
* Accessing an index of an array using [ and ].
* Creating empty arrays of a certain size, and filling the indices one by one.
* Getting the length of an array using length.
* Using the argument array args that is passed into the main() method of a class.

Lengte van een nieuwe array bepalen:

double[] mathScores = new double[4];

**Les 9 ARRAYLISTS**

Nodig om waarden te kunnen toevoegen en verwijderen in Java.

# **Introduction**

When we work with arrays in Java, we’ve been limited by the fact that once an array is created, it has a fixed size. We can’t add or remove elements.

But what if we needed to add to the book lists, newsfeeds, and other structures we were using arrays to represent?

To represent dynamic lists, we can use Java’s ArrayLists. ArrayLists allow us to:

* Store elements of the same type (just like arrays)
* Access elements by index (just like arrays)
* Add elements
* Remove elements

Remember how we had to import java.util.Arrays in order to use additional array methods? To use an ArrayList at all, we need to import them from Java’s util package as well:

import java.util.ArrayList;

Let’s learn how to make use of this powerful object…

# **Creating ArrayLists**

To create an ArrayList, we need to declare the type of objects it will hold, just as we do with arrays:

ArrayList<String> babyNames;

We use angle brackets < and > to declare the type of the ArrayList. These symbols are used for *generics*. Generics are a Java construct that allows us to define classes and objects as parameters of an ArrayList. For this reason, we can’t use primitive types in an ArrayList:

// This code won't compile:ArrayList<int> ages;// This code will compile:ArrayList<Integer> ages;

The <Integer> generic has to be used in an ArrayList instead. You can also use <Double> and <Char> for types you would normally declare as doubles or chars.

We can initialize to an empty ArrayList using the new keyword:

// Declaring:ArrayList<Integer> ages;// Initializing:ages = new ArrayList<Integer>();// Declaring and initializing in one line:ArrayList<String> babyNames = new ArrayList<String>();

# Zoekresultaten

## Samenvatting van internet

A **Java** both **int and Integer** are used to store **integer** type data the major **difference** between both is type of **int** is primitive while **Integer** is of class type. ... **int** helps in storing **integer** value into memory. **Integer** helps in converting **int** into object and to convert an object into **int** as per requirement.16 sep. 2019

# **Adding an Item**

Now we have an empty ArrayList, but how do we get it to store values?

ArrayList comes with an add() method that takes an argument to add to the end of the ArrayList:

# **ArrayList Size**

Let’s say we have an ArrayList that stores items in a user’s online shopping cart. As the user navigates through the site and adds items, their cart grows bigger and bigger.

If we wanted to display the number of items in the cart, we could find the size of it using the size() method:

ArrayList<String> shoppingCart = new ArrayList<String>();shoppingCart.add("Trench Coat");System.out.println(shoppingCart.size());// 1 is printedshoppingCart.add("Tweed Houndstooth Hat");System.out.println(shoppingCart.size());// 2 is printedshoppingCart.add("Magnifying Glass");System.out.println(shoppingCart.size());// 3 is printed

In dynamic objects like ArrayLists, it’s important to know how to access the amount of objects we have stored.

# **Accessing an Index**

With arrays, we can use bracket notation to access a value at a particular index:

double[] ratings = {3.2, 2.5, 1.7};System.out.println(ratings[1]);

This code prints 2.5, the value at index 1 of the array.

For ArrayLists, bracket notation won’t work. Instead, we use the method get() to access an index:

ArrayList<String> shoppingCart = new ArrayList<shoppingCart>();shoppingCart.add("Trench Coat");shoppingCart.add("Tweed Houndstooth Hat");shoppingCart.add("Magnifying Glass");System.out.println(shoppingCart.get(2));

This code prints "Magnifying Glass", which is the value at index 2 of the ArrayList.

# **Changing a Value**

When we were using arrays, we could rewrite entries by using bracket notation to reassign values:

String[] shoppingCart = {"Trench Coat", "Tweed Houndstooth Hat", "Magnifying Glass"};shoppingCart[0] = "Tweed Cape";// shoppingCart now holds ["Tweed Cape", "Tweed Houndstooth Hat", "Magnifying Glass"]

ArrayList has a slightly different way of doing this, using the set() method:

ArrayList<String> shoppingCart = new ArrayList<shoppingCart>();shoppingCart.add("Trench Coat");shoppingCart.add("Tweed Houndstooth Hat");shoppingCart.add("Magnifying Glass");shoppingCart.set(0, "Tweed Cape");// shoppingCart now holds ["Tweed Cape", "Tweed Houndstooth Hat", "Magnifying Glass"]

# **Removing an Item**

What if we wanted to get rid of an entry altogether? For arrays, we would have to make a completely new array without the value.

Luckily, ArrayLists allow us to remove an item by specifying the index to remove:

ArrayList<String> shoppingCart = new ArrayList<String>();shoppingCart.add("Trench Coat");shoppingCart.add("Tweed Houndstooth Hat");shoppingCart.add("Magnifying Glass");shoppingCart.remove(1);// shoppingCart now holds ["Trench Coat", "Magnifying Glass"]

We can also remove an item by specifying the value to remove:

ArrayList<String> shoppingCart = new ArrayList<String>();shoppingCart.add("Trench Coat");shoppingCart.add("Tweed Houndstooth Hat");shoppingCart.add("Magnifying Glass");shoppingCart.remove("Trench Coat");// shoppingCart now holds ["Tweed Houndstooth Hat", "Magnifying Glass"]

**Note:** This command removes the FIRST instance of the value "Trench Coat".

# **Getting an Item's Index**

What if we had a really large list and wanted to know the position of a certain element in it? For instance, what if we had an ArrayList detectives with the names of fictional detectives in chronological order, and we wanted to know what position "Fletcher" was.

// detectives holds ["Holmes", "Poirot", "Marple", "Spade", "Fletcher", "Conan", "Ramotswe"];System.out.println(detectives.indexOf("Fletcher"));

This code would print 4, since "Fletcher" is at index 4 of the detectives ArrayList.

# **Review**

Nice work! You now know the basics of ArrayLists including:

* Creating an ArrayList.
* Adding a new ArrayList item using add().
* Accessing the size of an ArrayList using size().
* Finding an item by index using get().
* Changing the value of an ArrayList item using set().
* Removing an item with a specific value using remove().
* Retrieving the index of an item with a specific value using indexOf().

Now if only there were some way to move through an array or ArrayList, item by item…

, "table tennis", "beach", "holidays", "frisbee", "tennis

Bij een arraylist hoef je bij uitprinten alleen maar het object (de instance) en met puntnotatie de arraylist te printen:

System.out.println((hobby.lol));

Weet nog niet wanneer ik moet compilen met:

**Javac classname.java**

En wanneer ik kan runnen met:

**Java classname**

En wanneer ik in Codeacademy op:

**Run**

Moet klikken

**Oefenen met de geleerde concepten**

import java.util.ArrayList;

import java.util.Arrays;

class List {

String[] activities;

ArrayList<String> lol = new ArrayList<String>();

public List(String[] activity, ArrayList<String> plezier) {

activities = activity;

lol = plezier;

}

public static void main(String[] args) {

System.out.println("hello world");

String[] weather = {"rain", "sun", "temperature", "clouds", "thunder", "moisture", "heat"};

String[] hobbies = {"soccer", "music", "tennis"};

System.out.println(Arrays.toString(weather));

System.out.println("test");

ArrayList<String> fun = new ArrayList<String>();

fun.add("soccer");

fun.add("tennis");

fun.add("pingpong");

fun.add("vrijworstelen");

System.out.println(fun.size());

List hobby = new List (hobbies, fun);

System.out.println("tesssst");

System.out.println((hobby.lol));

System.out.println((hobby.lol.get(1)));

fun.remove(3);

System.out.println((hobby.lol));

fun.set(2, "basketbal");

System.out.println((hobby.lol) + "derde element moet nu basketbal zijn");

System.out.println(fun.indexOf("tennis"));

}

}