CODEGYM

Mentor-supported training program from CodeGym

# Java Developer in 12 months

MODULE 1. JAVA SYNTAX

## Lesson 15 Lists

CODEGYM

# Lesson plan

- Wrapper types in Java
- Collections in Java: ArrayList
- Working with ArrayList
- Type inference by the Java compiler
- Parameterized types in Java: Generics
- A few facts about generics

CODEGYM

# Wrapper types in Java

**List of wrapper types**
Starting with the fifth version of Java, each primitive type gained a twin class. Each such class stores a single field with a value of a specific type.

These classes are called wrapper types, because they wrap primitive values in classes.

| Primitive type | Wrapper class |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| | |
| float | Float |
| double | Double |
| | |
| char | Character |
| boolean | Boolean |

The names of primitive types start with a lowercase letter, but the names of the wrapper classes start with an uppercase letter.

Please note:
**Integer** instead of **int** and **Character** instead of **char**.

# Autoboxing and unboxing

One of the features of Java's primitives and primitive wrapper classes is autoboxing/unboxing

**The automatic conversion of an int to an Integer is called autoboxing, and the reverse operation (converting an Integer to an int) is called unboxing.**

| Your code | What the compiler sees |
|---|---|
| Integer a = 10; | Integer a = Integer.valueOf(10); |
| int b = a; | int b = a.intValue(); |
| Integer c = a + b; | Integer c = Integer.valueOf(a.intValue() + b); |

CODEGYM

# Wrapper classes in detail

**Integer class**
The Integer class has two fields that contain the maximum and minimum possible values of the int type:

| Field | Description |
|---|---|
| Integer.MAX_VALUE | Maximum possible value of the int type |
| Integer.MIN_VALUE | Minimum possible value of the int type |

The Integer class also has some interesting methods. Here they are:

| Methods | Description |
|---|---|
| String Integer.toHexString(int) | Returns a string that is the hexadecimal representation of the number |
| String Integer.toBinaryString(int) | Returns a string that is the binary representation of the number |
| String Integer.toOctalString(int) | Returns a string that is the octal representation of the number |
| Integer Integer.valueOf(int i) | Wraps the passed int in an Integer object |
| Integer Integer.parseInt(String) | Returns the number obtained from the passed string |

CODEGYM

# Double class

## The Double class has six interesting fields:

| Field | Description |
|---|---|
| double Double.NEGATIVE_INFINITY | Negative infinity |
| double Double.POSITIVE_INFINITY | Positive infinity |
| int Double.MIN_EXPONENT | Minimum possible exponent ($2^x$) |
| int Double.MAX_EXPONENT | Maximum possible exponent ($2^x$) |
| double Double.MIN_VALUE | Minimum possible value of the double type |
| double Double.MAX_VALUE | Maximum possible value of the double type |

## And of course, the Double class has interesting methods:

| Methods | **Description** |
|---|---|
| String Double.toHexString(**double**) | Returns a string that is the hexadecimal representation of the number |
| **boolean** Double.isInfinite(**double**) | Checks whether the passed number is infinity. |
| **boolean** Double.isNaN(**double**) | Checks whether the passed number is **NaN** |
| Double Double.valueOf(**double**) | Wraps the passed double in a Double object |
| Double Double.parseDouble(String) | Returns the number obtained from the passed string |

CODEGYM

# Character class

**A feature of these methods is that they work with all known alphabets.**
**Useful methods:**

| Methods | Description |
|---|---|
| Character.isAlphabetic(int) | Checks whether a character is an alphabetic character |
| Character.isLetter(char) | Checks whether the character is a letter |
| Character.isDigit(char) | Checks whether the character is a digit |
| Character.isSpaceChar(char) | Checks whether the character is a space, a line break, or a page break (codes: 12, 13, 14) |
| Character.isWhitespace(char) | Checks whether the character is whitespace: a space, tab, etc. |
| Character.isLowerCase(char) | Checks whether the character is lowercase |
| Character.isUpperCase(char) | Checks whether the character is uppercase |
| Character.toLowerCase(char) | Converts the character to lowercase |
| Character.toUpperCase(char) | Converts the character to uppercase |

# Boolean class

The Boolean type has two constants (two fields):

| Constants of the class | Counterpart for the boolean type | Description |
|---|---|---|
| Boolean.TRUE | true | true |
| Boolean.FALSE | false | false |

# Collections in Java: ArrayList

**The name of the ArrayList class is formed from two words: Array + List.**
**Array is an array and List is a list.**

**The ArrayList class lacks the disadvantages that arrays have.**

It knows how to:

- Store elements of a specific type
- Dynamically resize the list
- Add elements to the end of the list
- Insert elements at the beginning or middle of the list
- Remove elements from anywhere in the list

CODEGYM

## Creating an ArrayList object

To create an ArrayList object, you need to write code like this:

```
ArrayList<TypeParameter> name = new ArrayList<TypeParameter>();
```

ArrayList is a collection type/class,
TypeParameter is the type of the elements stored in the ArrayList collection, and name is the name of an ArrayList<TypeParameter> variable.

| Code | Description |
|---|---|
| `ArrayList<Integer> list = new ArrayList<Integer>();` | List of integers |
| `ArrayList<String> list = new ArrayList<String>();` | List of strings |
| `ArrayList<Double> list = new ArrayList<Double>();` | List of real numbers |

Collections in Java, including ArrayList, can only store reference types.

CODEGYM

# Operations with an ArrayList

The numbering of the elements of the list starts from zero. The index of an element in a list may also be called the number of the element

| Methods | Description |
|---|---|
| void add(Type value) | Adds the passed element to the end of the list |
| void add(int index, Type value) | Adds an element to a specific location in the list. |
| Type get(int index) | Returns the element whose index is index |
| void set(int index, Type value) | Assigns value to the element whose index is index |
| Type remove(int index) | Removes the element whose index is index. Returns the removed element. |
| Type remove(Type value) | Removes the element that you pass to the method. If there is more than one such element, the first one will be removed. |
| void clear() | Clears the list, i.e. removes all elements from the list. |
| boolean contains(Type value) | Checks whether the list contains value. |
| boolean isEmpty() | Checks whether the list is empty or not. In other words, whether the length of the list is zero. |
| int size() | Returns the size of the list, i.e. the number of elements in the list. |
| Type[] toArray(Type[] array) | Returns an array containing the elements of the list. You need to pass the array to the method. |

# How ArrayList is structured

**The structure of ArrayList is simple and ingenious.  Each ArrayList object contains two fields:**

- An array of elements
- A size variable, which stores the number of elements in the list

**Internally, an ArrayList object contains a most ordinary array! But that's not all. There is also a size variable, which stores the length of the list.**

**Initially, the length of the array inside the list is 10. And the size variable is 0.**

**If you add another element when there is no more space in the array, then the following happens in the add() method:**

- A new array is created that is one and a half times the length of the previous one
- All the elements of the old array are copied into the new array.
- In the ArrayList object, a reference to the new array replaces the reference to the old one.
- The passed element is saved in the 10th cell of the new array.
- The size variable increases by 1 and will now equal 11

## Type inference by the Java compiler

**A slick language feature that lets you take a shortcut (write less code) is called syntactic sugar.**

### var

In Java 11, the compiler became even smarter and can now determine the **type of a declared variable based on the type of the value** assigned to it. In code, it looks like this:

```
var name = value;
```

The compiler itself determines, or infers, the variable's type based on the value assigned to it.

Type inference is useful with collections. Identical notation:

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

```
var list = new ArrayList<Integer>();
```

CODEGYM

# Omitting the type with the diamond operator: <>

Starting from the seventh version of Java, when writing a collection type, you could omit the type of the collection elements if it was specified when declaring a variable.

```
ArrayList<String> list = new ArrayList<>();
```

It's undesirable to use the var keyword and the diamond operator at the same time:

```
var list = new ArrayList<>();
```

There is no information at all about the type of the elements stored in the collection, and the collection type will be ArrayList<Object>.

CODEGYM

# Generics

**In Java, generics means the ability to add type parameters to types. The result is a complex composite type. The general view of such a composite type is this:**

```
ClassName<TypeParameter>
```

After the compiler compiles your generic classes with type parameters, they are simply converted to ordinary classes and typecast operators.

Information about the type arguments passed to variables of generic types is lost. This effect is also called type erasure.

CODEGYM

# A few facts about generics

Classes can have several type parameters.

```
ClassName<TypeParameter1, TypeParameter2, TypeParameter3>
```

Generic types can also be used as parameters.

```
ClassName<TypeParameter<TypeParameterParameter>>
```

Generic types (types with type parameters) can also be used as array types.

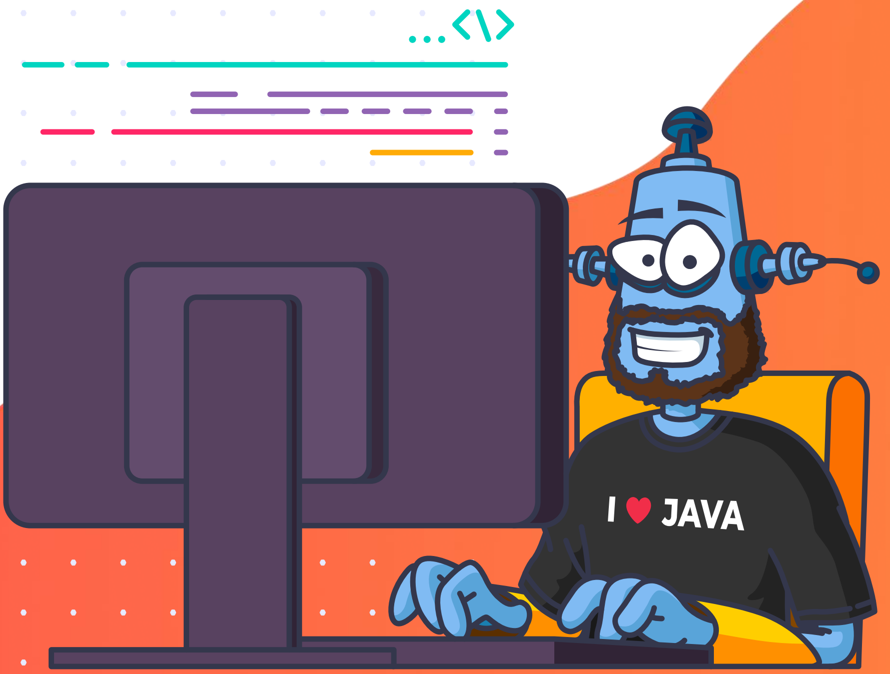
```
ClassName<TypeParameter>[] array = new ClassName<TypeParameter>[length];
```

CODEGYM

# Homework

MODULE 1. JAVA SYNTAX

Complete Level 16



MODULE 1. JAVA SYNTAX

CODEGYM

# Answers to questions

MODULE 1. JAVA SYNTAX