PROGRAMARE AVANSATĂ PE OBIECTE

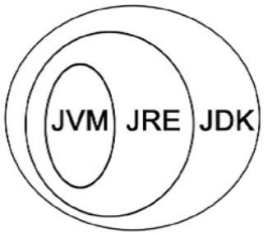
Laborator 1

Maria Cristina Chițu
mariachitu11@gmail.com

Contents

Short Introduction

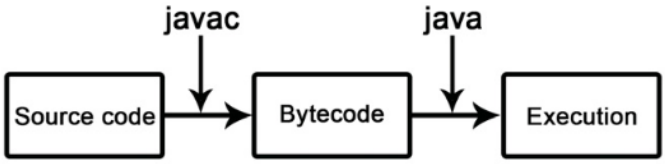
The three most basic parts of the Java ecosystem are the **Java Virtual Machine (JVM)**, the **Java Runtime Environment (JRE)**, and the **Java Development Kit (JDK).**



Every Java program runs under the control of a **JVM**. Every time you run a Java program, an instance of JVM is created. It provides security and isolation for the Java program that is running. It prevents the running of the code from clashing with other programs within the system.

Up in the hierarchy of stock Java technologies is the **JRE**. The JRE is a collection of programs that contains the JVM and also many libraries/class files that are needed for the execution of programs on the JVM (via the **java** command). It includes all the base Java classes (the runtime) as well as the libraries for interaction with the host system and utilities.

At the top layer of stock Java technologies is the **JDK**. The JDK contains all the programs that are needed to develop Java programs, and it's most important part is the Java Compiler (javac).

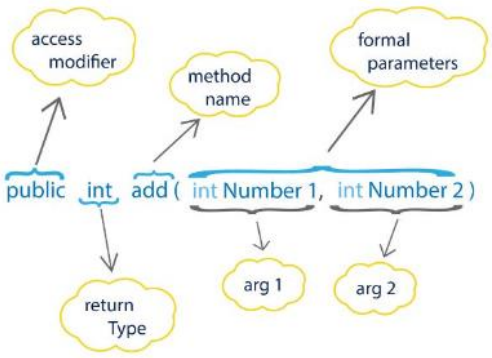


The process of compilation in Java

Java Class Structure

A Java class defines what objects of the class know (**fields**: hold the state of the program) and can do (**methods**: operate on the state of the program). The class also defines how to initialize the fields when the object is first created (**constructors**). Together they are called the **members of the class.**

The full declaration of a method is called a **method signature**.



- Most of the time, each Java class is defined in its own *.java file.
- You can even put two classes in the same file. When you do so, at most one of the classes in the file is allowed to be public.
- If you do have a public class, it needs to match the file name.

Ordering elements in a class:

Element	Example	Required?	Where does it go?
Package declaration	`package abc;`	No	First line in the file
Import statements	`import java.util.*;`	No	Immediately after the package
Class declaration	`public class C`	Yes	Immediately after the import
Field declarations	`int value;`	No	Anywhere inside a class
Method declarations	`void method()`	No	Anywhere inside a class

There are three different types of **comments**:

- Single-line comment: // comment until end of line
- Multiple-line comment:

```
/* Multiple
line comment
*/
```

- Javadoc comment:

```
/**
* Javadoc multiple-line comment
```

```
* @author mchitu
* @version 1.0
* @since 2020-02-27
*/
```

Packages (Declarations and Imports)

Packages are namespaces in Java that can be used to avoid name collisions when you have more than one class with the same name. It's necessary to organize our source files and group them by a criteria, such as functionality or some kind of relationship.

All the classes that are fundamental to the Java language belong to the **java.lang** package. All the classes that contain utility classes in Java, such as collections, classes for localization, and time utilities, belong to the **java.util** package.

As a programmer, you can create and use your own packages.

Here are a few rules to be considered while using packages:

- Packages are written in lowercase
- Package names should correspond to folder names.
- Import only classes (not methods, child packages, fields)

Wildcards: * is the wildcard that matches all the classes in the package. (example: import java.util.*;)

The package statement (for example, package graphics;) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

To use a class from a package in your code, you need to import the class at the top of your Java file. Import tells the compiler which package to look in to find a class. For example, to use the Student class, you would import it as follows:

```
import com.example.Student;

public class MyClass {

}
```

Naming conflicts: we use packages so that a class names doesn't have to be unique

across all of Java.

Example: Conflicts class (java.util.Date and java.sql.Date)

The first explicit import takes precedence over the second. (example: Conflicts class)

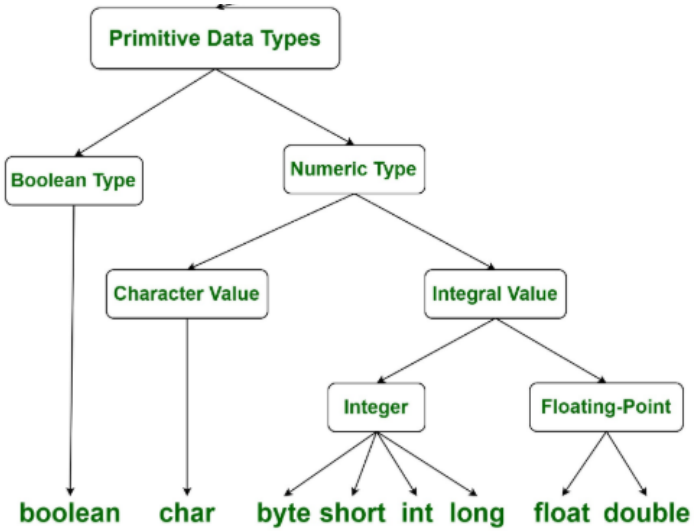
1. import java.util.Date;
2. import java.sql.Date; --Error: The import java.sql.Date collides with another import

statement.

Primitive types and wrapper classes

Primitive types are the fundamental types, that is, they cannot be modified. They are indivisible and form the basis for forming complex types. There are eight primitive data types in Java:

- byte
- short
- int
- long
- char
- float
- double
- boolean



TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\'','\n',' β'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

Generic classes only work with objects and don't support primitives. As a result, if we want to work with them, we have to convert primitive values into wrapper objects.

As the name suggests, wrapper classes are objects encapsulating primitive Java types.

Each Java primitive has a corresponding wrapper:

- boolean, byte, short, char, int, long, float, double
- Boolean, Byte, Short, Character, Integer, Long, Float, Double

These are all defined in the java.lang package, hence we don't need to import them manually.

Declaring and initializing variables

Whenever we want to deal with a given data type, we have to create a variable of that data type. For example, to create an integer that holds your age, you would use a line like the following:

int age;

The age variable now holds the value 30. The word age is called an **identifier** and is used to refer to the memory location where the value 30 is stored.

Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators - example: Conditional Operator (? :)

Instructions

Conditional Statements

- if-then

```
if (condition) {

//actions to be performed when the condition is true

}
```

- if-then-else

```
if (condition) {

//actions to be performed when the condition is true

}

else {

//actions to be performed when the condition is false

}
```

booleanExpression ? expression1 : expression2

- else-if

```
if (condition 1) {

//actions to be performed when condition 1 is true

}

else if (Condition 2) {

//actions to be performed when condition 2 is true

}

else if (Condition 3) {

//actions to be performed when condition 3 is true

}

…
```

- switch

To achieve this, we use a break statement to tell the compiler to continue to execute outside the switch statement. Here is the same switch case with break statements

Switch can be performed on the following data types:

- int and Integer;
- byte and Byte;
- short and Short;
- char and Character;
- String;
- enums;

```
switch (age){
case 10:
discount = 300;
break;
case 20:
discount = 200;
break;
case 30:
discount = 100;
break;
default:
discount = 50;
}
```

Looping statements

- while loops

```
while(condition) {
//Do something
}
```

- do-while loops

```
do {
//Do something
}
while(condition);
```

- for loops

```
for( initialization ; condition ; expression) {

//statements

}
```

- for each loops

```
for( type item : array_or_collection){

//Code to executed for each item in the array or
collection

}
```

Branching

- break
- continue
- return

Scanner Class in Java

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings.

Scanner is a class in java.util package used for obtaining the input.

For example, the packages use **nextInt()** to input an integer with the following syntax:

sc = new Scanner(System.in);

int x = sc.nextIn()

Example:

```java
package com.mariachitu.reading;

import java.util.Scanner;

public class ReadInput {
    public static void main(String[] args) {
        // Java program to read data of various types using Scanner class.
        // Declare the object and initialize with
        // predefined standard input object

        Scanner sc = new Scanner(System.in);

        // String input
        System.out.println("Enter your name: ");
        String name = sc.nextLine();

        // Character input
        System.out.println("Enter your gender: ");
        char gender = sc.next().charAt(0);

        // Numerical data input
        // byte, short and float can be read
        // using similar-named functions.
        System.out.println("Enter your age: ");
        int age = sc.nextInt();
        System.out.println("Enter your phone number: ");
        long mobileNo = sc.nextLong();
        System.out.println("Enter your GPA: ");
        double grade = sc.nextDouble();

        // Print the values to check if the input was correctly obtained.
        System.out.println("Name: "+name);
        System.out.println("Gender: "+gender);
        System.out.println("Age: "+age);
        System.out.println("Mobile Number: "+mobileNo);
        System.out.println("GPA: "+grade);

    }
}
```