Object oriented programming

Java

Part 1: Introduction

History – versions (1)

- **[1995] Version 1.0**, called ***Java Development Kit*** **(JDK)**
  - 212 classes in 8 packages
  - Slow, lot of *bugs*, but already with *Applets*

- **[1997] Version 1.1**, called JDK 1.1
  - 504 classes in 23 packages
  - Improvements in the efficiency of the JVM
  - Main extensions: nested classes, JavaBeans, Java Database Connectivity (JDBC), Java Remote Method Invocation (Java RMI), ...

- **[1998] Version 1.2**, henceforward called ***Java 2 Platform*** **(J2SE)**
  - 1520 classes in 59 packages
  - Sun Java Virtual Machine (JVM) with Just In Time (JIT) compiler
  - Main extensions: Swing, collections, ...
  - Code name *Playground*

History – versions (2)

- **[2000] Version 1.3**, called J2SE 1.3
  - 1842 classes in 76 packages
  - Improvements in the efficiency of the JVM
  - Code name *Kestrel*

- **[2002] Version 1.4**, called J2SE 1.4
  - 2291 classes in 135 packages
  - Improvements in the efficiency of the JVM
  - Main extensions: assertions, exceptions, security and cryptography, ...
  - Available in three platforms:
    - Java 2 Micro Edition (J2ME), for mobiles and PDAs
    - Java 2 Standard Edition (J2SE), for desktops
    - Java 2 Enterprise Edition (J2EE), for enterprises

History – versions (3)

  - Comprehensive facilities to computer programmers provided by IDEs (*Integrated Development Environment*)
    - NetBeans, from Sun
    - Eclipse, from IBM
  - Code name *Merlin*

- **[2004] Version 5.0**, called J2SE 5.0
  - 3000 classes in 165 packages
  - Main extensions: generics, enums, primitive types and wrapper classes, variable number of arguments, ...
  - **Version previously numbered 1.5**
  - Code name *Tiger*

- **[2006] Version 6.0**, calles J2SE 6.0
  - Main extensions: Extensible Markup Language (XML), web services, …
  - Code name *Mustang*

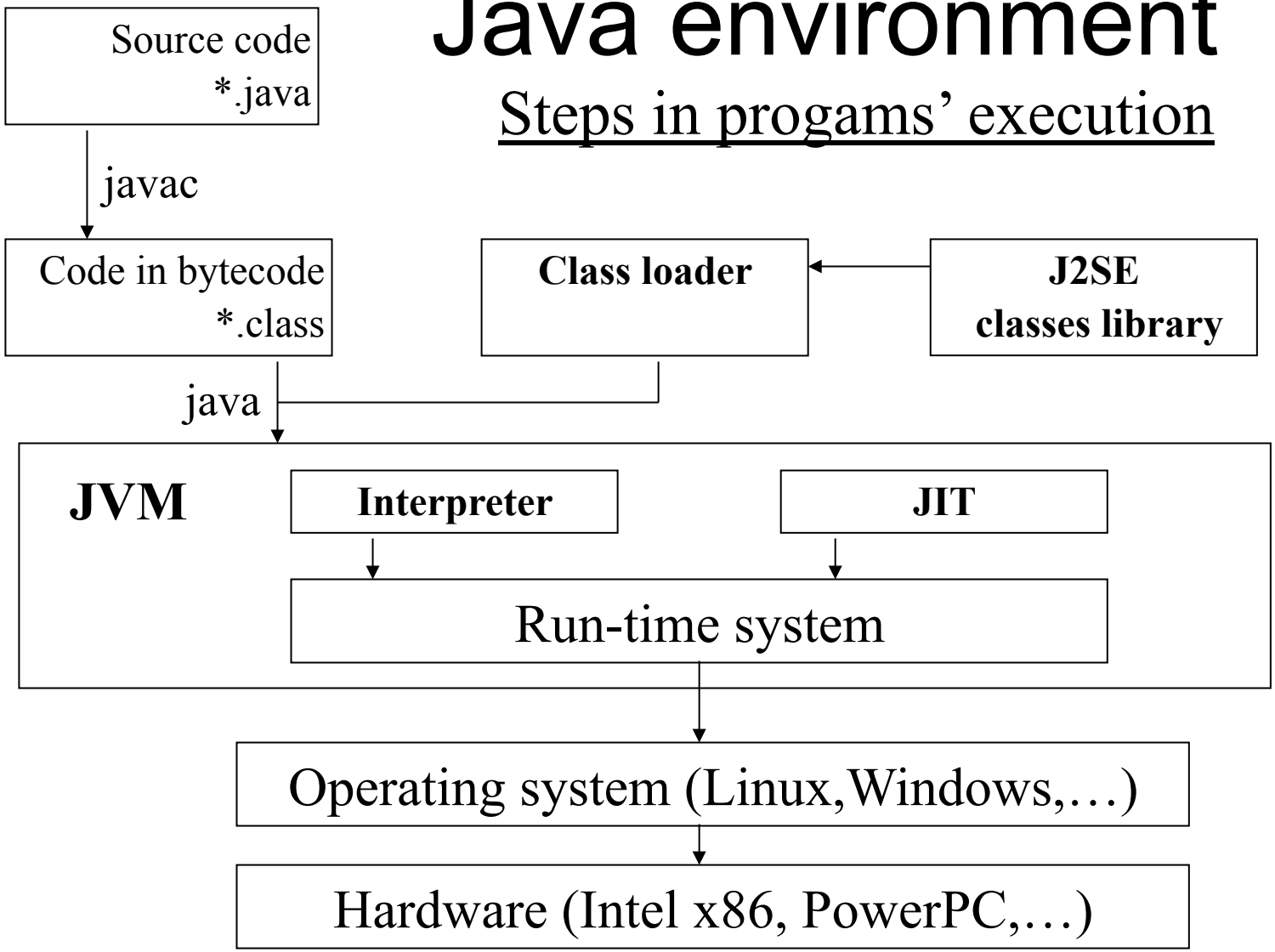
History– versions (4)

- **Versions J2SE 7** has code name *Dolphin*, and it was released in 2011.
  - JVM with support for dynamic languages
  - Improvements in the *garbage collector*
  - Main extensions: parallel computations in multi-core processors, super packages, ...

- **Version J2SE 8**, released in 2014!

Java environment

Steps in progams' execution



Java platform (1)

- Java technology is distributed for 3 platforms:
  - J2EE (*Enterprise Edition*), for the development of enterprise applications.
  - J2ME (*Micro Edition*), for devices with limited capacities (mobiles and PDA's).
  - J2SE (*Standard Edition*), for desktops and servers.

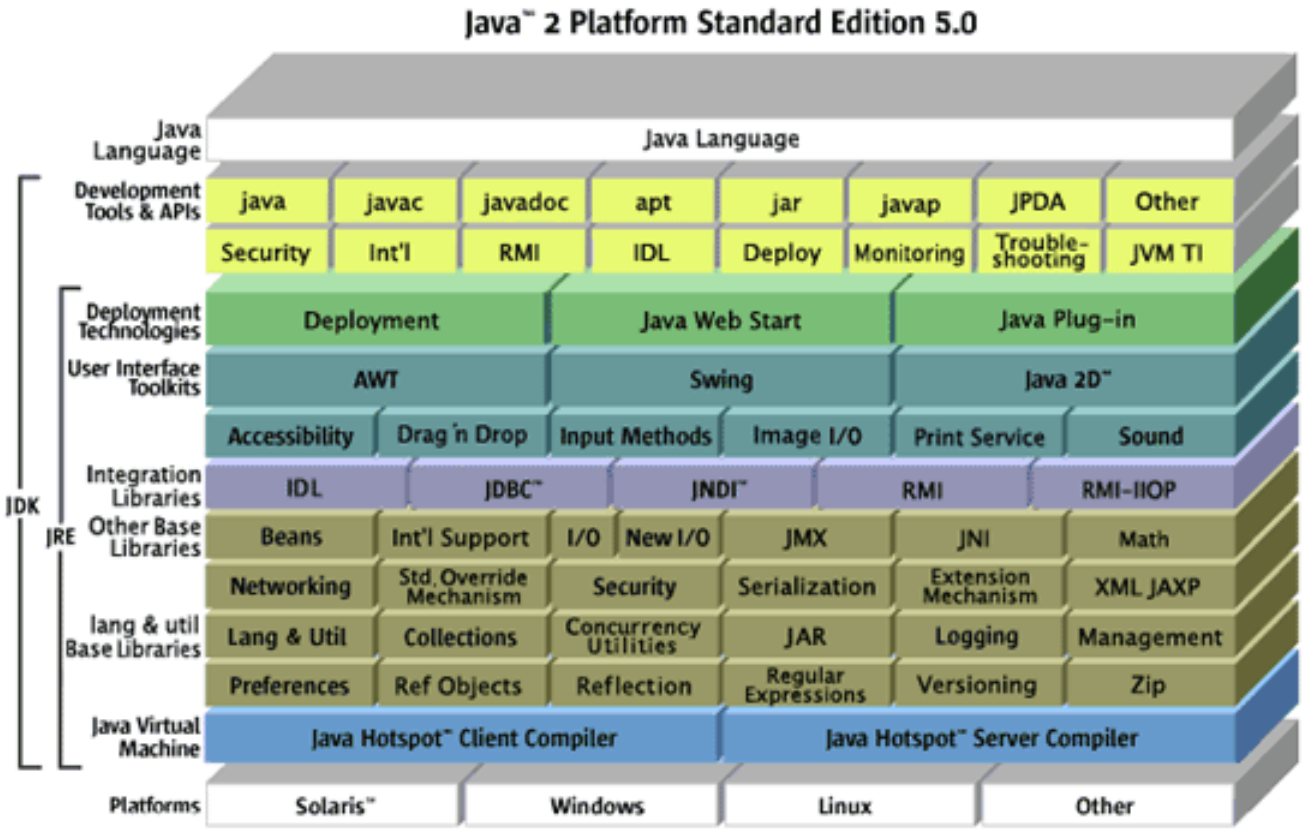
Java platform (2)

- Inside this platform there are different directories:
  - J2xx Runtime Environment (JRE):
    - Interpreter JVM, environment classes, …
    - Used (only) to run applications.
  - J2xx Development Kit (JDK):
    - JRE, compiler, utility classes (Swing,…), …
    - Used to develop applications.

Java platform (3)

- Java API consists in different classes distributed and organized in packages and sub-packages.
- Basic packages:
  - **java.lang**: environment classes (automatically imported)
  - **java.util**: utility classes (data types, etc)
  - **java.io**: I/O classes
  - java.net: classes for network usage (TCP/IP)
  - java.sql: classes to access via JDBC
  - java.awt: native graphical interface
  - **javax.swing**: graphical interface (lighter than java.awt)
- Documentation:
  - https://docs.oracle.com/javase/8/docs/api/

Java platform (4)



Java language

- Language particularities, comparatively to C/C++:
  - There are no pointers; instead there are **references**.
  - There is **`new`**, but not **`free`**/**`delete`** (***garbage collector***).
  - The **parameters are passed by value** to the methods.
  - There isn't *operator overloading*.
  - There are no multiple inheritance of classes, only of interfaces.
  - There are no *preprocessor* nor *header files*.
  - There are no global variables.
  - It is **strongly typed**.
  - Variables can be declared in any place inside the method, and not only in the beginning.
  - There are no **`goto`**, **`typedef`**, **`union`**, or **`struct`**.
  - There may exist more than one **`main`** (but only one per class).

Java: references (1)

- Java doesn't use pointers:
  - There are references, which indeed are **implicit pointers.**
  - **There is no pointer arithmetic**, the implicit pointers are never explicitly used as in C/C++.
  - They are treated as any other ordinary variable.
  - Every object in Java is found in the heap.

Java: references (2)

- Java primitive types (**`char`**, **`int`**, **`long`**, etc) are treated differently from objects:
  - **Primitive types**:

```
int iVar;
```

    - Integer variable called **`iVar`**.
    - The actual value of the variable is stored in a memory address called **`iVar`**.
    - Before any assignment it stores a default value: 0.

Java: references (3)

- **Objects**:

```
BankAccount baVar;
```

  - **baVar** is a **reference to an object** of type **BankAccount**.
  - The memory address called **baVar** does not store the object itself, but a reference to an object of that type; the object is stored elsewhere in memory.
  - Before any assignment it stores a reference to a special object: **null**.

Java: **new** operator

- Any object in Java must be created using the **new** operator:

```
BanckAccount baVar1;
baVar1 = new BankAccount();
```

  - The **new** returns a reference (not a pointer).
    - The programmer doesn't know the object memory address.
  - It's not necessary to free memory.
    - Java verifies periodically every block of memory allocated with a **new** to check if there is still a valid reference to it (***garbage collector***).
    - Avoids ***memory leaks***.

Java: assignment

- When assigning references, two references to the same object exist:

```
BankAccount baVar1, baVar2;
baVar1 = new BankAccount();
baVar2 = baVar1;
```

  - Both variables are references to the same object.
  - If on both variables an withdraw of 1000€ is done then, in the end, the bank account in question will have less 2000€ than initially.

Java: equality/identity(1)

- **Primitive types**: **==**
  - The equality operator (==) tell us whether two variables have the same value, as in C/C++.

```
int iVar1 = 27;
int iVar2 = iVar1;
if (iVar1==iVar2)
   System.out.println("They're equal!");
```

Java: equality/identity(2)

- **Objects: identity** with **==**

```
BankAccount baVar1 = new BankAccount();
BankAccount baVar2 = baVar1;
if (baVar1==baVar2)
   System.out.println("They're identical!");
```

  - The equality operator (==), when applied to objects, tell us whether two references are identical. That is, whether they refer to the same object.

Java: identity/equality (3)

- **Objects: identity** with **`equals`**

```
BankAccount baVar1 = new BankAccount();
BankAccount baVar2 = new BankAccount();
if (baVar1.equals(baVar2))
 System.out.println("They're equal!");
```

  - The method **`equals`** is related with identity of objects, that is, to check whether two objects have the same data/state.
  - By default the method **`equals`** returns the same as the operator ==, but it should be redefined if identity between object is needed.

Java: parameters

- In Java, parameters are always passed by value.
  - The object is never copied, only the reference is copied, referring both to the same object.

```
void method1() {
   BankAcccount baVar = new BankAccount();
   method2(baVar);
}
void method2(BankAccount baArg) {}
```

  - Both references **baVar** and **baArg** refer to the same object.
- In C/C++ arguments are passed by value, but the object is copied too. If this is not desired a pointer to an object must be used.
  - In C++ **baArg** would be a new object, copied from **baVar**.

Java: input/output (1)

- Output:
  - Any primitive type (numbers and chars), as well as **`String`** type objects, is printed in the following form:

```
System.out.print(var);
System.out.println(var);
```

    - The **`print`** method prints the value of **`var`**.
    - The **`println`** method prints the value of **`var`** and terminates the current line.
    - Variables/literals may also be separated by the **`+`** operator:

```
System.ou.println("The answer is " + var);
```

      - The result would be: "The answer is 15"
        (is the value of **`var`** is 15).

Java: input/output (2)

- Input:
  - It is mandatory to import in the begging of the java source file:

```
import java.io.*;
```

  - Abnormal situations may occur, for instance, a file not found, therefore reading methods typically throw exceptions of type **`IOException`**.
  - From the input stream an object of type **`String`** is read. If one needs to read any another type, for instance, a character or a number, it is necessary to convert the **`String`** into the desired type.

Java: input/output (3)

- **Reading a `String` from the keyboard**:

```
public static String getString() throws IOException {
        InputStreamReader isr = new
        InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s = br.readLine();
        return s;
}
```

Java: input/output (4)

- **Reading a `char` from the keyboard**:

```
public static char getChar() throws IOException {
        String s = getString();
        return s.charAt(0);
}
```

  - The **`charAt`** method returns the corresponding character from the object **`String`**.
  - In this example the first character is returned (at index 0).

Java: input/output (5)

- **Reading an `int`/`long` from the keyboard**:

```
public static int getInt() throws IOException {
        String s = getString();
        return Integer.parseInt(s);
}

public static long getLong() throws IOException {
        String s = getString();
        return Long.parseLong(s);
}
```

  - The **`parseInt`**/**`parseLong`** method from class **`Integer`**/**`Long`**, converts an object of type **`String`** into an **`int`**/**`long`**.

Java: input/output (6)

- **Reading a `double`/`float` from the keyboard**:

```
public static int getDouble() throws IOException {
        String s = getString();
        Double d = Double.valueOf(s);
        return d.doubleValue();
}
public static float getFloat() throws IOException {
        String s = getString();
        Float f = Float.valueOf(s);
        return f.floatValue();
}
```

  - The method **`ValueOf`**, of class **`Double`**/**`Float`**, converts an object of type **`String`** into an object of type **`Double`**/**`Float`**.
  - The method **`doubleValue`**/**`floatValue`**, of class **`Double`**/**`Float`**, converts an object of type **`Double/Float`** into **`double`**/**`float`**.

Java: input/output (7)

There is also the class Scanner, useful for breaking down formatted input into tokens and translating individual tokens according to their data type:

```
import java.util.Scanner;
…
Scanner scan = null;
try{
        scan = new Scanner(System.in);
        System.out.print("Enter student number:");
        int nb = scan.nextInt();
        scan.nextLine(); //the \n is not read from nextInt
        System.out.print("Enter student name:");
        String name = scan.nextLine();
        System.out.println("Student\n -nb: "+nb+"\n -name: "+name);
} // catch …
```

Java: main

- All classes in a Java application may have a **`main`** method.

```
public class HelloWorld {
        public static void main(String[] args) {
                System.out.println("Hello, world!");
        }
}
```

- Each Java source file should contain only one public class, and the name of the file must be exactly the name of the class with extension .java.
- The *Java virtual machine* (JVM) interpreter executes the **main method** of the class indicated in the command line.

```
> javac HelloWorld.java
> java HelloWorld
> Hello world!
```