OOSD

Introduction to JAVA

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

September 12, 2011

Outline

JAVA

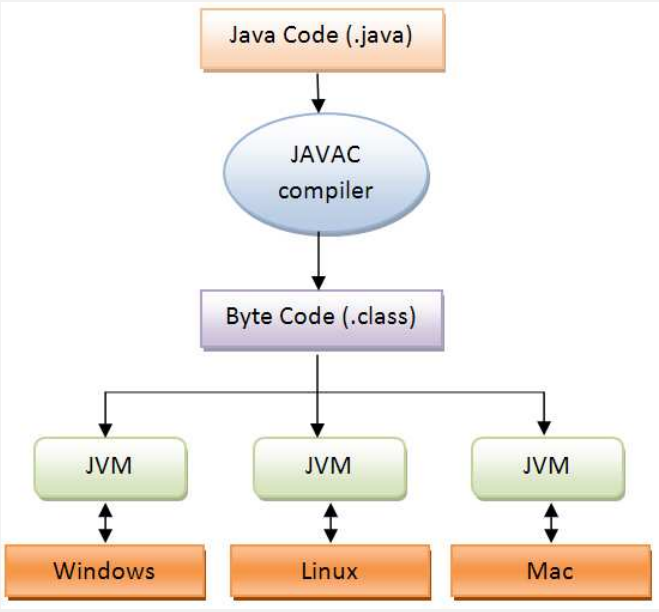
JAVA is a language developed by James Gosling at Sun Microsystems in 1995.

- Initially developed as an internal project, it was specifically designed for embedded systems (sic!)
- it later became very popular for general purpouse programming
- The most prominent features of JAVA are
  - **Portability:** JAVA programs are compiled into Bytecode, that can be executed by the **Java Virtual Machine (JVM)**
  - **Easy of programming:** JAVA is considered a programming language that favors clean and structured code, and avoids some of the most difficult aspects of other programming languages

JVM

JAVA is *semi-compiled*

- A JAVA program is first compiled into *bytecode*
  - The bytecode is an intermediate and platform independent low-level representation of the program
  - It is portable, i.e. it is independent of the underlying hardware processor architecture
- The bytecode is *interpreted* and executed by the JAVA VIRTUAL MACHINE
- The JAVA RUN-TIME ENVIRONMENT (JRE) includes the JVM and supporting library classes



Bytecode

Of course, this approach has advantages and disadvantages:

- **Portability** is the greatest advantage: the same compiled bytecode can be executed on any OS and on any hardware platform
- Other advantages: Dynamic loading, run-time type identification, etc.
- **Efficiency** is the primary concern: bytecode is not as efficient as native machine code, as it needs to be interpreted by the JVM that is a regular program.
  - This issue is partially mitigated by the introduction of **Just-In-Time compilers**: they translate byte-code into machine code *on-the-fly*, while executing, thus speeding up the execution of programs

Which language is faster?

- Comparing the *performance* of different languages is quite difficult, if not impossible because it depends on:
  - The algorithm
  - The programmer ability
  - The input
  - The Operating System and the JVM,
  - etc.
- In general, it is possible to say that JAVA programs performance is[1]:
  - 1-4 times slower than compiled languages as C or C++
  - close to other Just-in-time compiled languages such as C♯,
  - much higher than languages without an effective native-code compiler (JIT or AOT), such as Perl, Ruby, PHP and Python.
- Also, see `http://bit.ly/9AwPXG` for a comparison of JAVA against various other languages

[1] See `http://en.wikipedia.org/wiki/Java_performance`

First JAVA program

A simple *Hello world* in JAVA HelloDate.java

```
// HelloDate.java
import java.util.*;

public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

Tell the system to import all classes in an external package (see Date below)

Program starts from here

Prints the string on the screen

Prints the date on the screen

Classes

- the `class` keyword define a class
  - everything is an object: therefore, every JAVA program only contains classes declarations and objects
  - every class goes in a separate file that must have the same name as the class name (in our case, `HelloDate.java`)
  - every JAVA program must contain a class with a method called `main`
- By convention, usually all class names begin with a capital letter, to distinguish them from variables that start with a lower letter.

Exercise

Compile and execute the program

A more interactive example

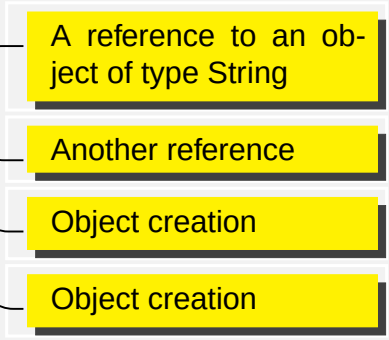
IntroducingHello.java

```
import java.util.*;

class IntroducingHello {
  public static void main(String[] args) {
    String name;
    String surname;

    name = new String("Giuseppe");
    surname = new String ("Lipari");

    System.out.println("Hello " + name
                       + " " + surname);
    System.out.println("Today is: ");
    System.out.println(new Date());
  }
}
```



- There is no object yet, only references!
- Now the first object is created

Objects and references

- In JAVA, you treat everything as an object, using a single consistent syntax
- The identifier you manipulate is actually a "reference" to an object
  - You might imagine this scene as a television (the object) with your remote control (the reference).
  - As long as you're holding this reference, you have a connection to the television, but when someone says "change the channel" or "lower the volume" what you're manipulating is the reference, which in turn modifies the object.
  - If you want to move around the room and still control the television, you take the remote/reference with you, not the television.
  - Also, the remote control can stand on its own, with no television. That is, just because you have a reference doesn't mean there's necessarily an object connected to it.

References and object creation

When you create a reference, you want to connect it with a new object. You do so, in general, with the new keyword. The keyword new says, “Make me a new one of these objects.” So in the preceding example, you can say:

```
String s;
s = new String("Giuseppe");
```

- `s` is the reference (the remote control)
- initially, every reference is automatically initialized to **`null`**, a special constant to denote “nothing”. `s` is a dangling reference, it refers to nothing
- You create an object with the **new** keyword, followed by the class name and the initialization parameter

References

- `String` is a predefined class in JAVA, but we will see how to do it with our own classes
- You can define a reference and initalize it in the same line: (the latter is only valid for `String`)

```
String s = new String("Giuseppe");
String s2 = "Lipari";
```

Primitive types

"Everything is an object" has exceptions.

- basic low level data are implemented as primitive types. Here are the primitive types in java.

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	–	–	–	Boolean
char	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8-bit	-128	$+127$	Byte
short	16-bit	-2^{15}	$+2^{15}-1$	Short
int	32-bit	-2^{31}	$+2^{31}-1$	Integer
long	64-bit	-2^{63}	$+2^{63}-1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	–	–	–	Void

- The size of the boolean type is not explicitly specified; it is only defined to be able to take the literal values true or false.

Primitive types

Primitive type variables are not object

- You can define and assign a value to a primitive variable, without using the **new** command

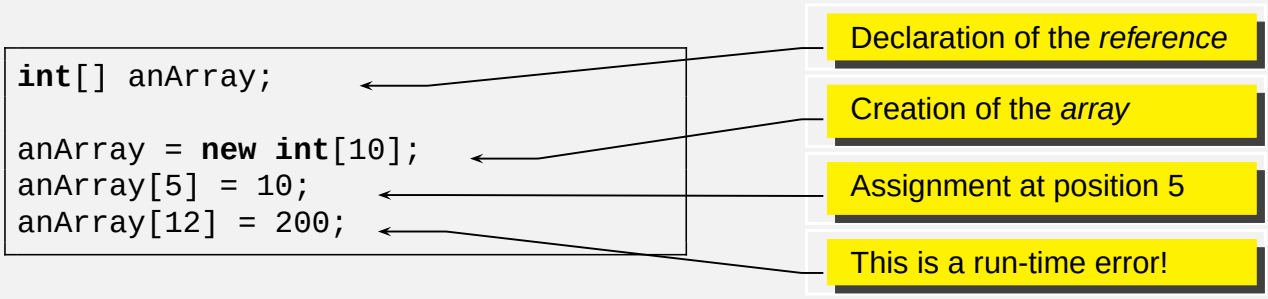
```
char c = 'x';
int i = 0;
int a, b;
a = b = 5;
int c = a + b;
```

- Wrappers are objects that "wrap" primitive types.

```
Integer a = new Integer(5);
Integer b = Integer.valueOf("125");
```

Arrays

- Arrays objects that contain sequences of other elements
- Unlike other containers (`Lists`, `Trees`, etc.), which are implemented as pure objects into libraries, arrays are objects built-in into the language



- Notice that the array index goes from 0 to 9, thus 10 is an invalid index.

Arrays - II

Arrays can also be initialized inline:

```
int[] anArray = {100, 200, 300, 400, 500,
                 600, 700, 800, 900, 1000};
```

Multi-arrays are arrays of arrays. Here is an example:

```
class MultiDimArrayDemo {
  public static void main(String[] args) {
    String[][] names = {{"Mr. ", "Mrs. ", "Ms. "}, {"Smith",
                        "Jones"}};
    System.out.println(names[0][0] + names[1][0]); //Mr. Smith
    System.out.println(names[0][2] + names[1][1]); //Ms. Jones
  }
}
```

You can also know the number of elements in an array:

```
System.out.println(anArray.length);
```

Copying arrays

Be careful when copying objects! WrongArrayCopy.java

```
import java.util.*;

class WrongArrayCopy {
    public static void main(String [] args) {
        String [] mys = {"Giuseppe", "Lipari"};
        String [] mys2 = mys;

        mys2[0] = "Roberto";
        System.out.println("mys[0] = " + mys[0]);
        System.out.println("mys2[0] = " + mys2[0]);
    }
};
```

- The problem is that in this way we are only copying the reference, not the entire object!
- You must create a new object, then copy elements one by one
  - The JAVA library helps us with an efficient method for copying arrays
  - see `./examples/03.java-examples/ArrayCopyDemo.java`

The underlying platform

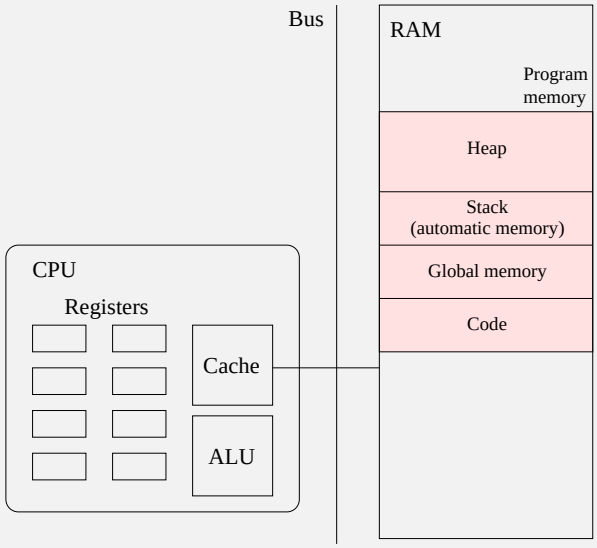
- JAVA is a language that is closer to the problem domain than lower level languages like C (which are closer to the machine domain)
- However, at some point JAVA has to be translate on a machine
- It is important then to understand *the mapping* between the high level abstractions of JAVA and the underlying low level platform details
- In facts, to understand how a JAVA program works, you must understand how the underlying machine translates and executes every statement
- In other words, you must be able to *be the machine*

Object creation

What happens when an object is created?

- First of all, memory is reserved for holding the object data

Registers: very small amount of memory in the processor, controlled by the compiler, you cannot see it
Cache: small amount of memory, automatically managed at run-time, you cannot see it
Stack: in the RAM, contains all local (automatic) variables inside a function
Heap: in the RAM, contains all new objects created dynamically by the programmer (you!)



Object creation

When JAVA creates an object, it goes in the Heap.

- You can think of the heap as an organized archive,
  - everytime you create an object it goes in the archive, and the bookkeeping system of JAVA gives you a *handle* to the object (the reference)
- You can never directly use the object, but only use the reference to it as a *remote control*.
- The bookkeeping system of JAVA keeps track of where objects are in the archive, so that when you want to use the object (through the reference), it knows where it is
- However, if you lose the reference (for some reason), you lose control of the object!
- There is no way to control an object without a reference to it

Stack

Primitive types are created on the stack, at the beginning of the execution of a function

- Every time you call a function (method) of a class, the stack is filled with the parameters and the local variables
- The stack is cleaned up when the function is finished, and all the variables on the stack are deleted
- When you call **new**, instead, object are created on the heap and will not be deleted after the end of the function, but will stay until an appropriate time

Object lifetime

- In most programming languages, the concept of the lifetime of a variable occupies a significant portion of the programming effort.
  - How long does the variable last?
  - If you are supposed to destroy it, when should you?
- Confusion over variable lifetimes can lead to a lot of bugs,
- JAVA greatly simplifies the issue by doing all the cleanup work for you

Scope

- In C, C++, and Java, scope is determined by the placement of curly braces {}. So for example:

```
{
  int x = 12;
  // Only x available
  {
    int q = 96;
    // Both x & q available
  }
  // Only x available
  // q "out of scope"
}
```


Scope - II

- Note that you cannot do the following, even though it is legal in C and C++:

```
{
  int x = 12;
  {
    int x = 96; // Illegal
  }
}
```

- Thus the C and C++ ability to "hide" a variable in a larger scope is not allowed, because the Java designers thought that it led to confusing programs.

Object lifetime

- JAVA objects do not have the same lifetimes as primitives. When you create a Java object using new, it hangs around past the end of the scope.

```
{
  String n;
  {
    String s = new String("a string");
    n = s;
  } // End of scope
  System.out.println(n);
}
```

- the reference `s` vanishes at the end of the scope, however, the `String` object is still occupying memory.

Garbage collector

- If JAVA leaves the objects lying around, what keeps them from filling up memory and halting your program?
  - JAVA has a *garbage collector* (GC), which looks at all the objects that were created with **new** and figures out which ones are not being referenced anymore
  - Then, it releases the memory for those objects, so the memory can be used for new objects
  - This means that you never need to worry about reclaiming memory yourself.
  - It eliminates a certain class of programming problem: the so-called "memory leak," in which a programmer forgets to release memory.

GC drawbacks

- The GC is executed periodically, or when the amount of objects goes beyond a certain threshold, or with some other rule
- Of course, not deleting objects immediately means that
  - objects stay in memory longer than strictly needed, so JAVA programs tend to use more memory than necessary
  - the GC execution can take a lot of computational resources
- Will come back to GC later.

Classes

- In procedural languages like C and C++, code is written inside functions and procedures, and a special function `main()` is the program entry point that calls all other functions
- Where goes the code inside JAVA?
- In JAVA, everything is an object of some type, hence the job of the programmer is principally to:
  - Write new types where to define data, and operations (functions) that operate on the data (see Abstract Data Type lecture)
  - Write a special class which contains the special entry point function `main()`
- Hence, all the programming activity in JAVA starts with designing the types, and implement them

Classes

In JAVA and in other OO languages, to define a new type we must use the keywork **`class`**

```
class ATypeName { /* Class body goes here */ }
```

- This introduces a new type, even if the class body is empty
- The following class define a different type:

```
class AnotherTypeName { /* Class body goes here */ }
```

- to do something useful, we must be able to write data and code for the class

Fields and methods

- A class can contain:
  - **Fields** (or *data members*) which are variables that contain data (primitive variables) or references to objects
  - **Methods** (or *member functions*) which are mostly operations on the object fields
- fields are used to specify the domain of interest
- methods are used to specify the operations on the object
- We will now analyse the syntax. Later we will look at simple design strategies

Data fields

The syntax for specifying a data field is the following (brackets denote optional elements)

```
[access] [scope] type name [ = initial_value];
```

- **access** can be *public*, *protected* or *private* (default is public for the package, will see later what it means)
  - If *public*, the field is part of the interface and can be accessed directly
  - If *private*, the field is part of the implementation and can be accessed only by other member functions of the same class
  - If *protected*, the field is part of the implementation and can be accessed only by other member functions of the same class or of derived classes

Scope and Type

- **scope** can be omitted or be the keyword *static*
  - In the first case, there is a separate and distinct copy of the field in every object of that type
  - In the second case, there is only one copy of the field, the same for all object of the same class
- **type** is the field type
  - Can be a primitive type or the name of another class
- **name** is the name of the field
- The initial value is optional (default is null for references, 0 for numeric primitive types, **`false`** for boolean)

Member access

```
class DataOnly {
  String n;
  int i;
  float f;
  boolean b;
}
...
DataOnly d = new DataOnly();
d.i = 47;
d.f = 1.1f;
d.b = false;
d.n = "object name";
System.out.println(d.n);
```

by default these fields are public

Since they are public, you can access them directly in the code

- Members can be accessed with the *dot notation*, i.e. the name of the reference followed by a dot and the name of the member (field of method)

Methods

Methods are operations on the object.

- their syntax is similar to functions in C

```
returnType methodName( /* Argument list */ ) {
  /* Method body */
}
```

- The return type is the type of the value that "pops out" of the method after you call it
- The argument list gives the types and names for the information you want to pass into the method
- The method name and argument list together uniquely identify the method

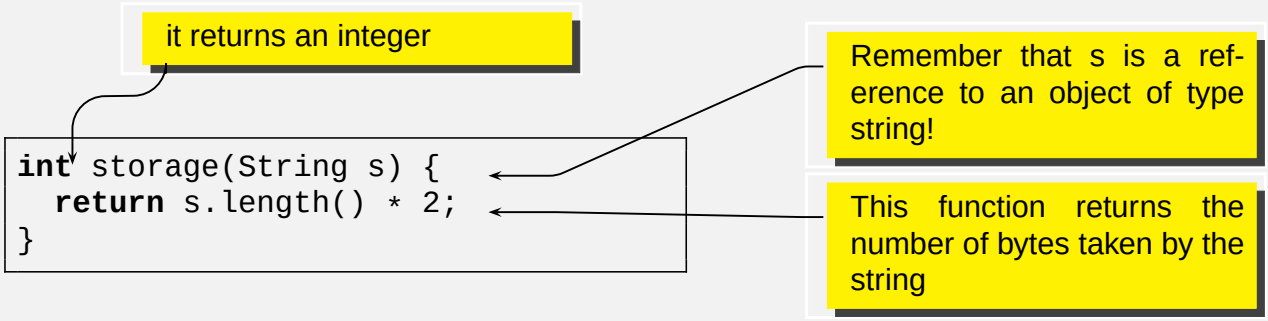
Methods in JAVA can be created only as part of a class

- A method can be called only for an object (except for static methods), like this:

```
objectName.methodName(arg1, arg2, arg3);
```

Arguments

- The argument list specifies what information you pass into the method
- For every argument, you must specify the type



Exercise

Try to *mentally* execute the code of the function when it is called on a certain string. Then write a program with a class the contains the method and call it from the main function.

Return types

- The **return** keyword means two things:
  - "leave the method, I'm done"
  - if the method produces a value, that value is placed right after the return statement
- You can return any type you want, if you don't want to return anything at all, you do so by indicating that the method returns **void**

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
void nothing2() {}
```

Compiling

To compile a JAVA program, you can use the JAVA compiler `javac`

- Code is written in text files with extension .java (e.g. `./examples/03.java-examples/HelloDate.java`)
- The compiler takes a .java file and produces a .class file
  - `javac HelloDate.java` produces `HelloDate.class`
  - The .class file contains the **bytecode**
- To execute the program, we must invoke the JAVA virtual machine, passing the name of the class file to interpret (without the .class)
  - `java HelloDate`

Name visibility

- Name clashing problem:
  - If you use a name in one module of the program,
  - and another programmer uses the same name in another module,
  - how do you distinguish one name from another and prevent the two names from "clashing?"
- JAVA solves the issue by using Internet domain names to specify the "location" of objects in the libraries
  - the retis library of utility objects will be at `it.sssup.retis.utility.*`
  - The domain name is in reverse order, and the directories are separated by dots
- Each file contains at least one class
  - the name of the file must be the same as the name of the class (except for the extension .java)
  - For example, class HelloDate must be in file HelloDate.java

The import keyword

- To use a class `MyClass`, you have the following possibilities:
  - If the class is in the same file where it is used, you can just use it (e.g. create objects of that class)
  - If the class is in another file in the same directory (Package), again you can just use it
  - If the class is in another place, you have to *import* the class with the import keyword
- Example of import:

```
import java.util.Date;  // imports class Date
import java.util.*;     // imports all classes from
                        // java.util package
```

- Notice that package `java.lang` is implicitely included

Documentation for JDK

- The Java Development Kit (JDK) from Sun comes with a full set of libraries
- Documentation for these libraries can be found at `http://download.oracle.com/javase/6/docs/api/`
  - For example, here is the documentation for Date: `http://download.oracle.com/javase/6/docs/api/java/util/Date.html`

Static keyword

- Objects have identity
  - Every time you create an object of one class, one instance of a complete object is created with its own member fields

```
class MyClass {
  int x;
  int y;
}
...
MyClass obj1 = new MyClass;
MyClass obj2 = new MyClass;

obj1.x = 5;
obj2.x = 7;
```

These two are different variables, as they belong to different objects

Static

- If you specify a member as *static*, it means that there will be **only one copy** of that member shared by all objects of that class

StaticExample.java

```
class MyClass {
    int x = 0;
    int y = 0;
    static int w = 0;
}

class StaticExample {
    public static void main(String args[]) {
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();

        obj1.x = 5;
        obj2.x = 7;
        obj1.w = 10;

        System.out.println("MyClass.w = " +
                           MyClass.w);
```

These are regular members

This member is shared between all objects of MyClass

Static methods

- A static member exists even without creating an object of the class
  - In the previous example, you can use member `w` before creating `obj1` and `obj2`.
- A method declared *static* can only access static members
  - The static method is not *tied* to a specific instance of the class (object), so it can only act on static variables
- Why static members and methods?
  - One of the most popular uses is to count how many objects of a specific class have been created
    - every time an object is created, the counter is incremented (will see later)
  - Another use is to implement functions that have nothing to do with one specific object
  - Another use is to control object creation more finely

Example of static method

StaticFun.java

```
class MyClass {
    private int x = 0;
    private static int counter = 0;

    public static int incr() {
        return counter++;
    }
    public MyClass() {
        x = incr();
    }
    public int getX() { return x; }
}

class StaticFun {
    public static void main(String args[]) {
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();

        System.out.println("obj1.getX() = " +
                           obj1.getX());
        System.out.println("obj2.getX() = " +
                           obj2.getX());
    }
}
```

A static method

Constructor

"Getter" method

Comments

- There are two types of comments in JAVA
  - The first is the traditional C-style comment that was inherited by C++
    - they start with /* and end with */
    - Often, if spread across many lines, a * is put at the beginning of each line
  - The second is one-line comments starting with //, again inherited from C++

```
// this is a one-line comment

/* this is a multiple-line
very long comment */

/* this one is nicely
 * indented, and can be nicer to
 * see on screen */
```


Comments for documentation

- To document the classes interface, you can use special comments:
  - They start with /** and end with */
  - Or, they start with ///
- These comments can be extracted by *javadoc*, an utility that generates html files

```
/** A class comment */
public class DocTest {
  /** A variable comment */
  public int i;
  /** A method comment */
  public void f() {}
}
```

Documentation

- Inside special comments, it is possible to use HTML (to a limited extent)
- Also, special keywords for cross referencing

This is a complete example by Bruce Eckel:
`./examples/03.java-examples/HelloDateDoc.java`

Exercises

Exercise

Write a program that prints three arguments taken from the command line. To do this, you'll need to index into the command-line array of Strings.

Exercise

Write and compile a program that tries to use a non-initialized reference. See what happens when you execute it.