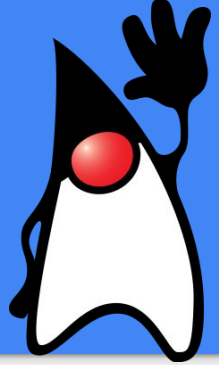
Java Training

Day 1: Overview of the language & “Hello World”

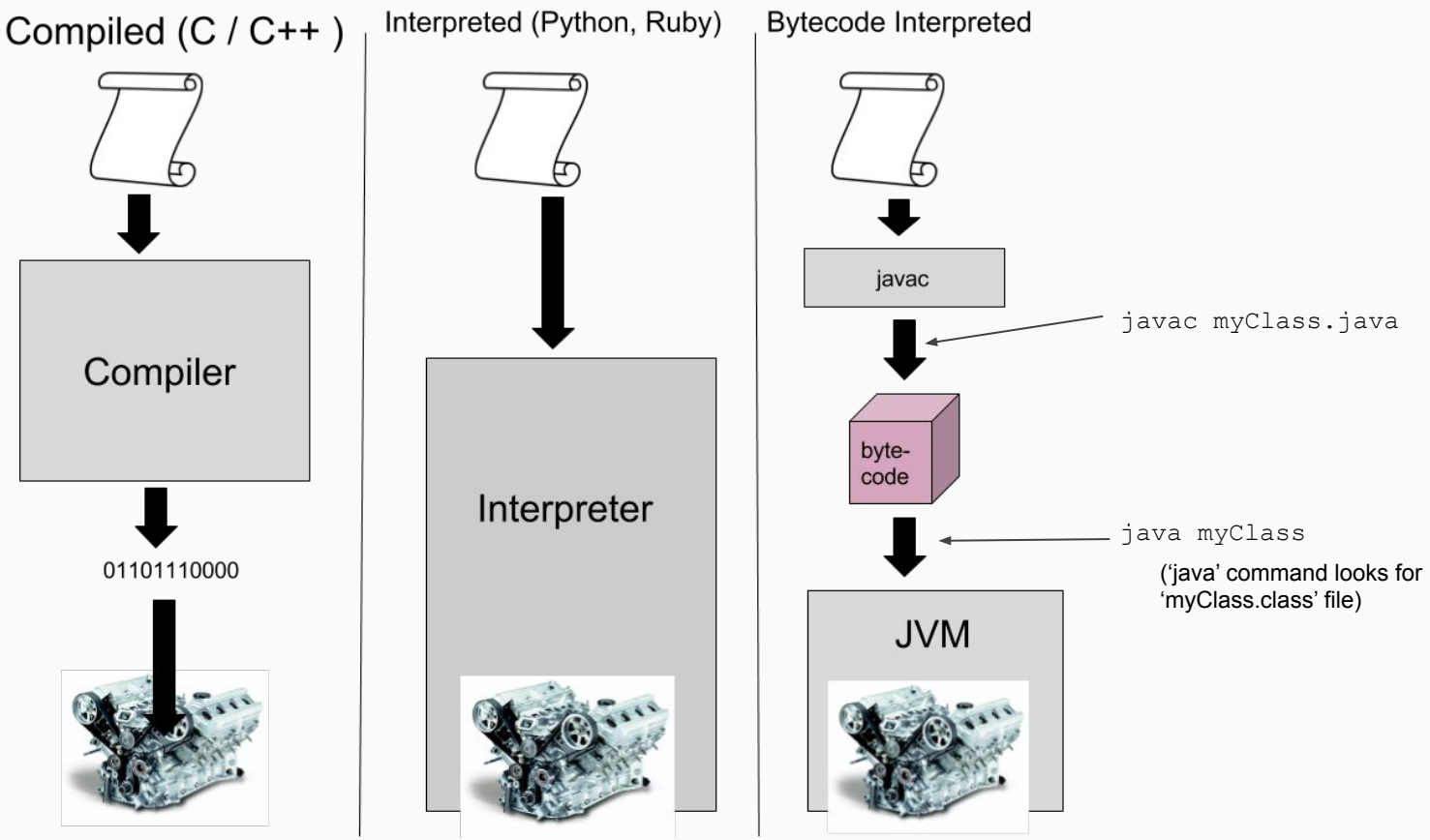
Download today’s slides:
go/java+espresso-training/day1

The Java Programming Language



- Designed James Gosling at Sun Microsystems, 1995
- Syntactically similar to C/C++
- Object-Oriented (Class based)
- Bytecode interpreted, “Write once, run anywhere!”
- Most popular programming language in the world (according to the TIOBE index for 2017)

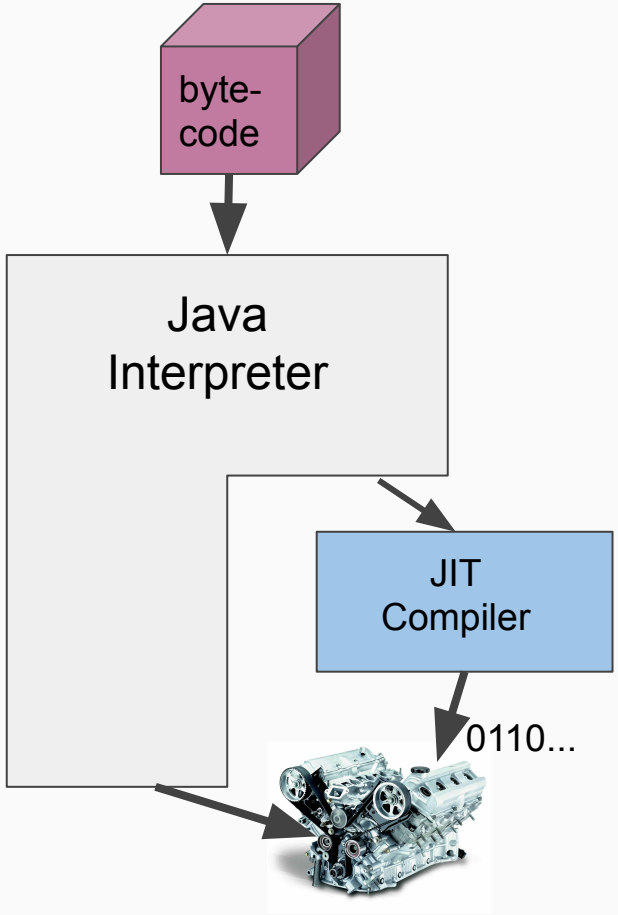
What does byte-code interpreted mean?



Innards of the JVM

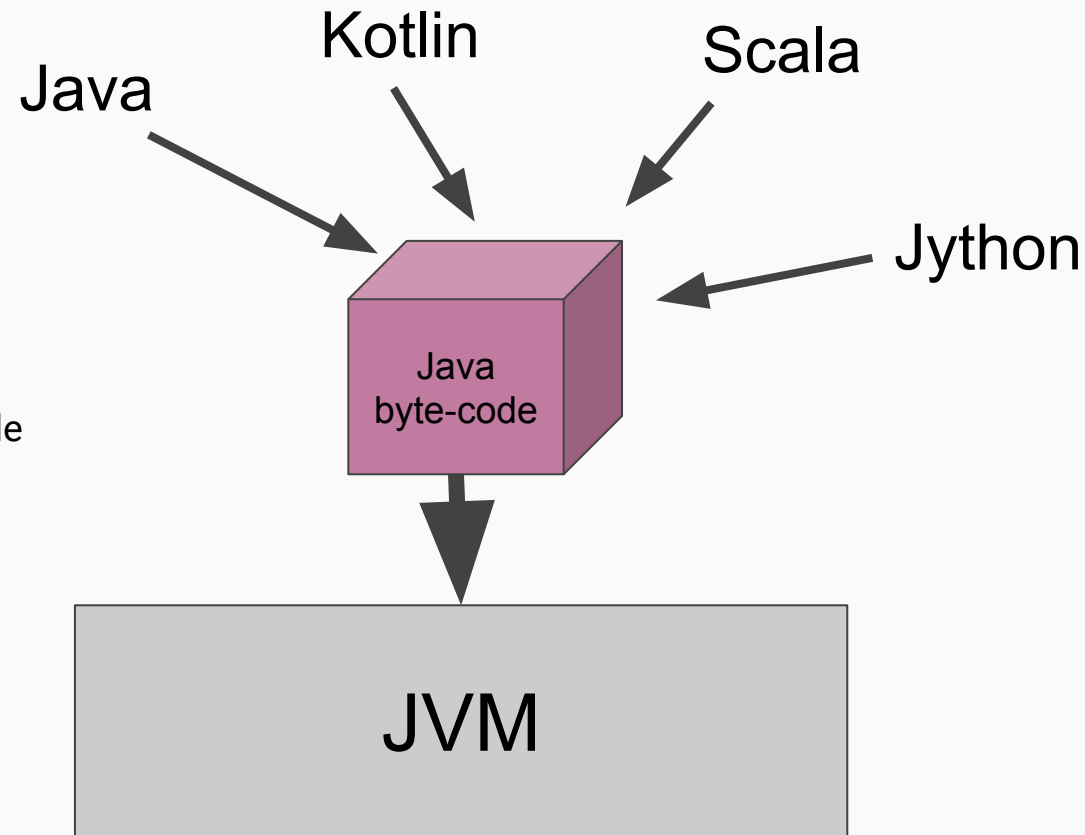
Interpreting (even if it's byte-code) is still slow(one layer of indirection) compared to compiled code, but execution tends to be repetitive: the same loops and methods get run over and over

The JVM optimizes by compiling these code *hotspots* and executing them as machine-code directly on the processor



Java Byte-code as an intermediate language representation

Another useful feature of Java byte-code is that it is a *language-independent* intermediate code representation

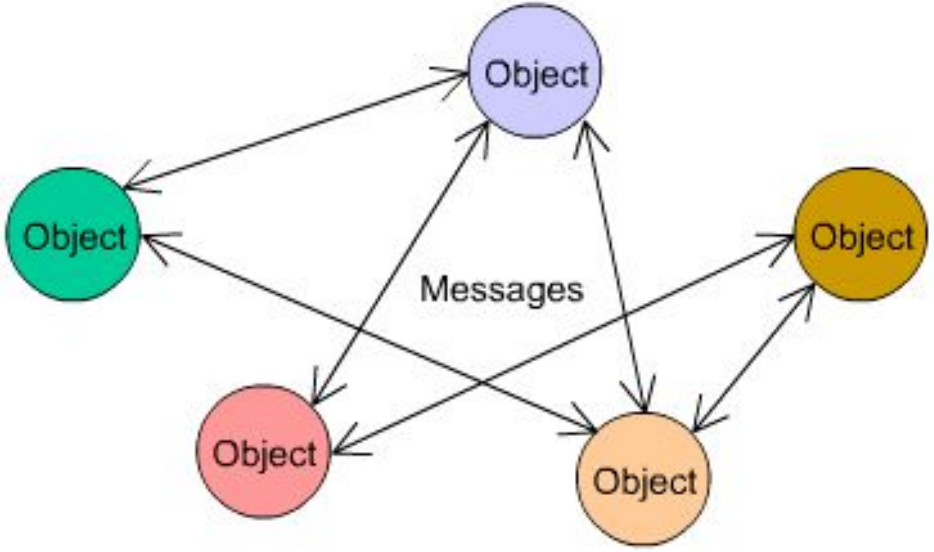


Byte-code classes compiled from different JVM compatible languages may be combined and run on the JVM

Object Oriented Programming

- Unlike *Procedural Programming* where data and functions and data are decoupled, *Object Oriented Programming* ties them together within *objects*.
- You *can* practice OO coupling in nearly any language (e.g. C structures with function pointers), but language support makes life much easier
- There were earlier languages with OO support, *Smalltalk* (Xerox PARC in the 70s) really made *Object Oriented Programming* a computing paradigm

The Smalltalk “message-passing” model

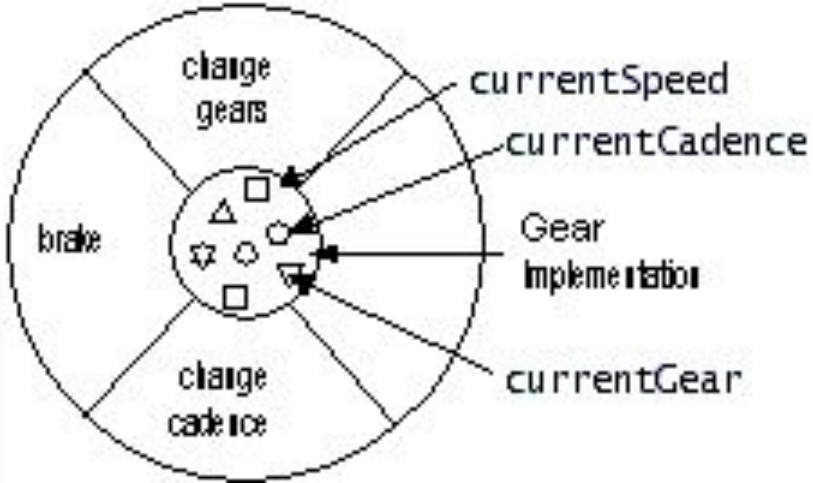


Interaction of objects via message passing

Objects have data(internal state) and know how to perform a set of actions.

They perform an action when they receive a message to do so.

Bicycle example:

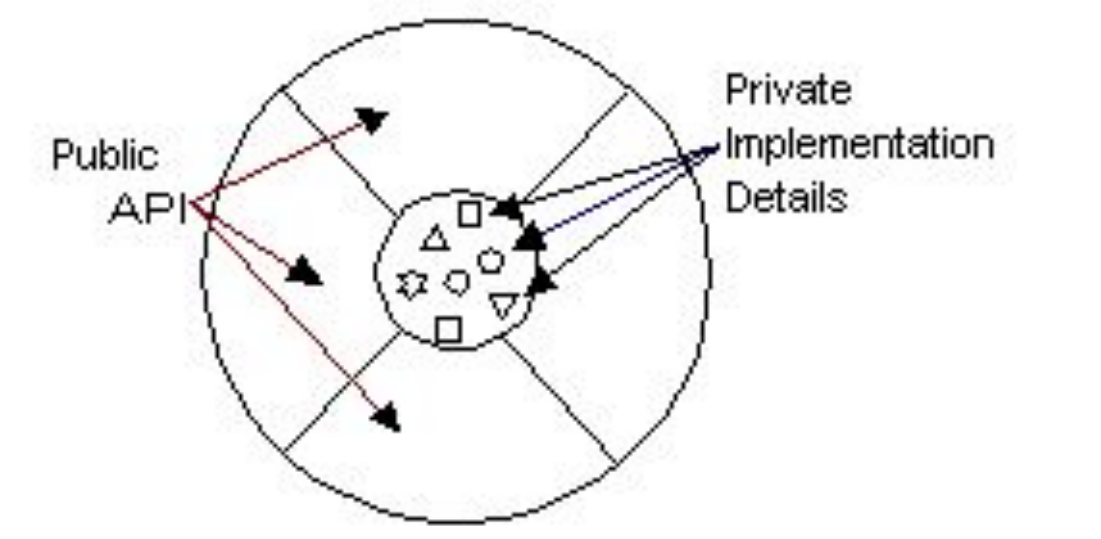


Message-passing, part 2

- **The Public-API** defines the types of messages the object will respond to. These might include messages allowing the reading or modifying of data (state).

- **Private Implementation Details** are data or methods the object keeps to itself.

A private method defines a message type the object is willing to accept from itself, but not from the outside world

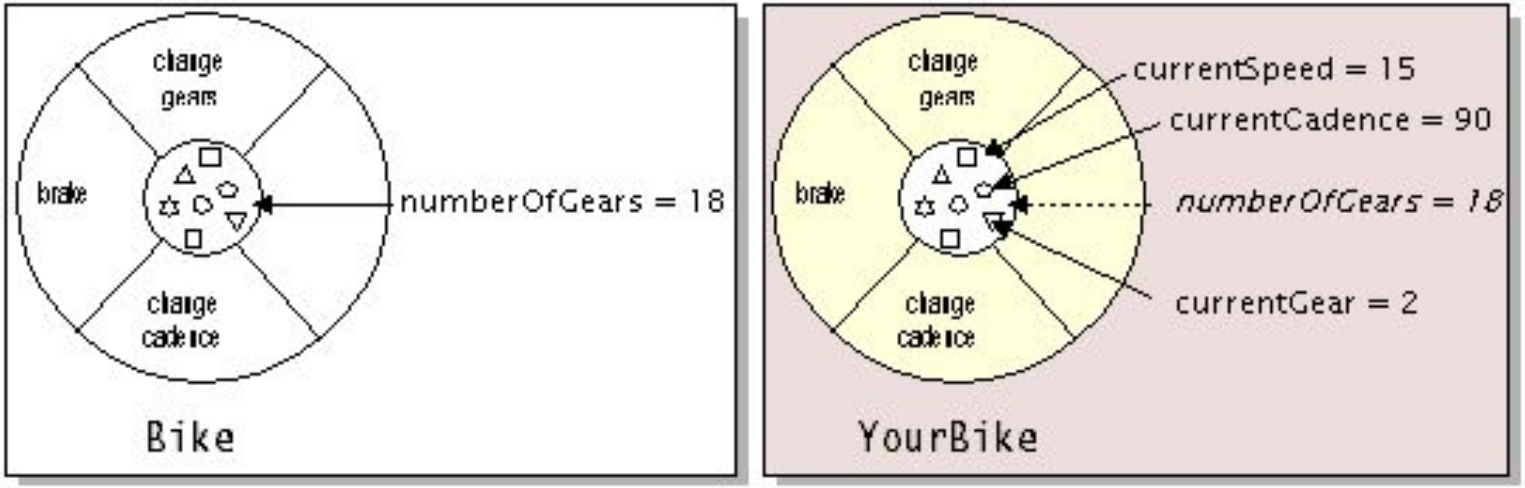


Classes and Instances

In Smalltalk the relationship between *classes* and *objects* is straightforward. Classes are merely *templates* or *blueprints* and all objects are instances of a class.

In Smalltalk you can send a message to an *instance* of the class Bike (such as **YourBike**) but sending a message to the **Bike** class is meaningless since it only exists as a *template* for creating *instances*.

In more recent OO languages (Java, Swift, C++, Objective-C) things are a bit weirder....

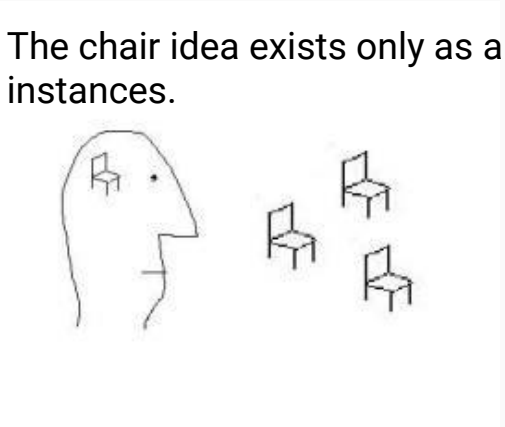


Class **Instance of a Class**

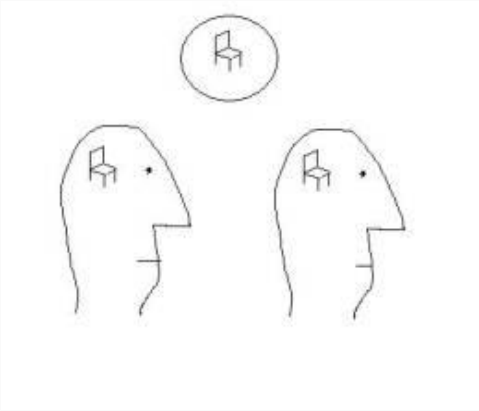
Java Classes and Plato's theory of Forms

Objects that are instances of a class are a bit like chairs made from an idea of a chair in a craftsman's imagination.

The chair idea exists only as a template for making physical chair instances.



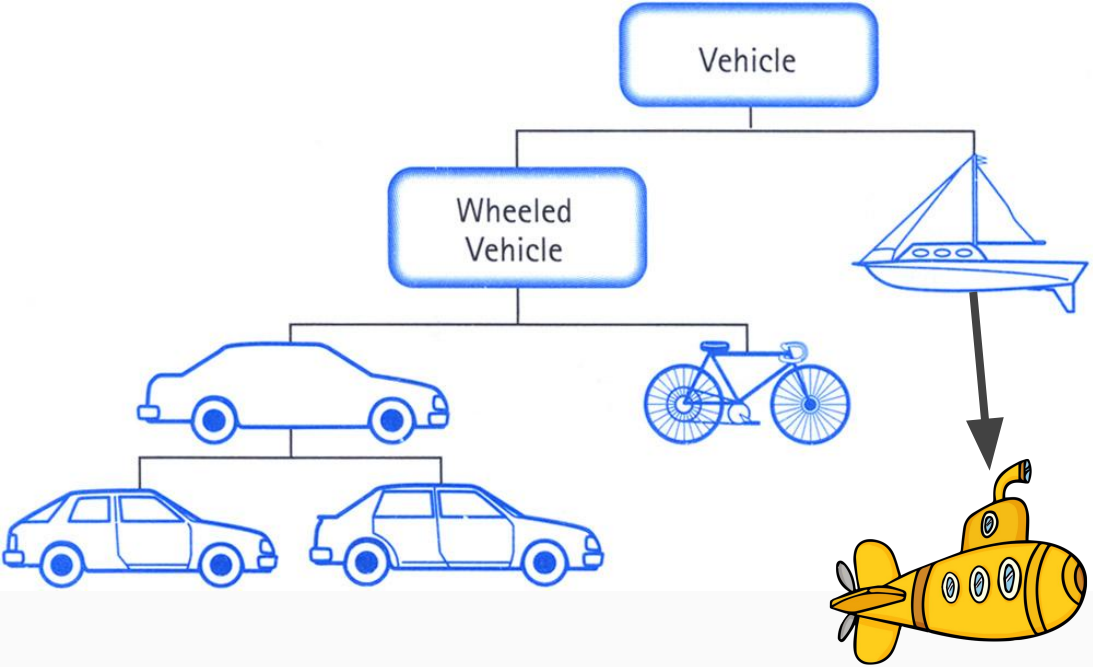
Plato had an idea of "Forms" where in the ideal of the chair had a metaphysical existence that informed every chairmaker's idea of a chair.

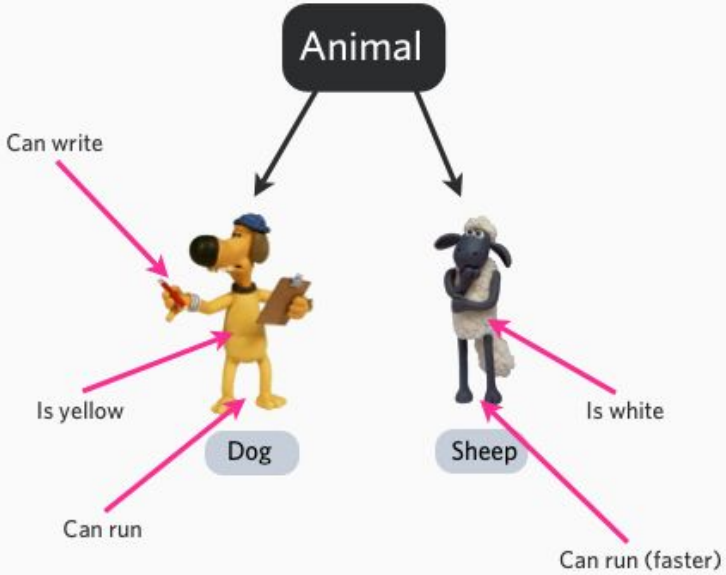


A **class object** is like a Platonic form. It is a template for creating instance objects, but also exists independently as its own unique object, and is also shared between instance objects

An Object Oriented perspective is a good fit for a lot of problems

Coupling functions(methods) with data fits with the notion of **Actors** that perform **Actions** and have **Attributes**





The “Hello World!” program

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

In the terminal we’ll be using two commands to compile and run java code:

`javac HelloWorld.java -` compiles your java language class into bytecode

`java HelloWorld -` passes your bytecode to the JVM for execution

Dissecting ‘HelloWorld’

`public` makes this a publicly available symbol.
Other options: `private` or `protected`

Class name - the name doesn’t matter as long as first class the JVM loads has a void() method where program execution can start

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

`static` means the method is associated with the class object rather than an instance of the class

String array of command-line arguments to the program

```
> java HelloWorld lions tigers bears
```

The variable `args` will contain: “lions”, “tigers”, “bears”

More Dissecting ‘HelloWorld’

String argument to the method

```
System.out.println("Hello world.");
```

Class members and methods are also lower-case

Class names are capitals, while instances and methods are lower-case.
This is why we created the `HelloWorld` class and not the `helloWorld` class.

Knowing this, we can *infer* that `System` is class object rather than and instance that we’re calling a method on (actually we’re calling a method on `out`, which is a static data member(instance) of type `PrintStream`

The `System` class looks like this

```
public final class System {
    static PrintStream out;
    static PrintStream err;
    static InputStream in;
    ...
}

public class PrintStream extends FilterOutputStream {
    //out object is inherited from FilterOutputStream class
    public void println() {
    ...
}
```