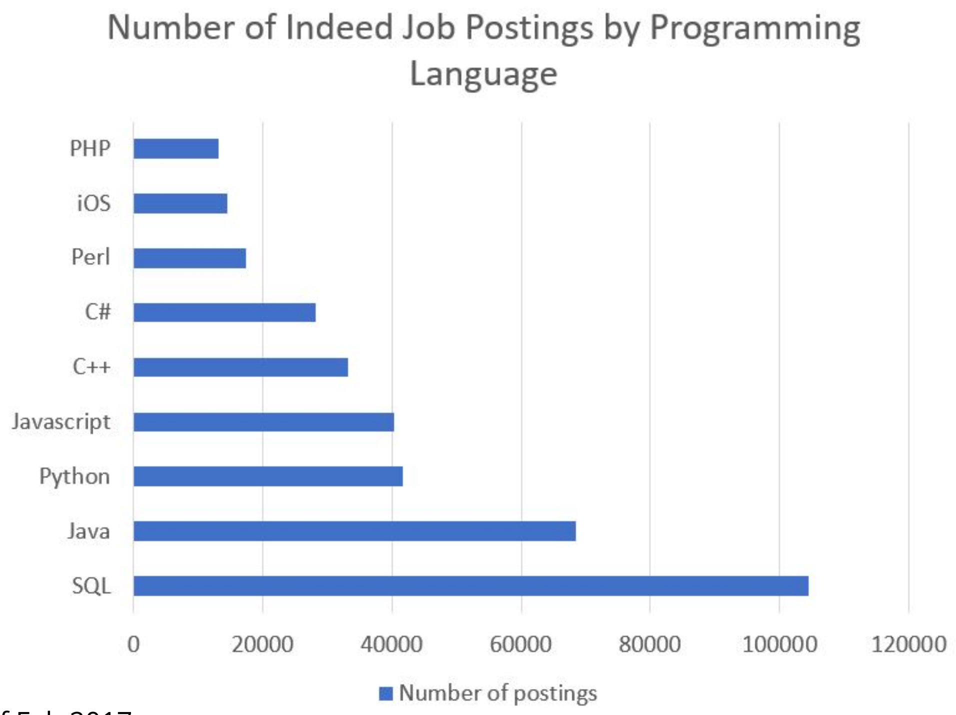
Java Basics

Language

Inheritance

Interfaces



As of Feb 2017

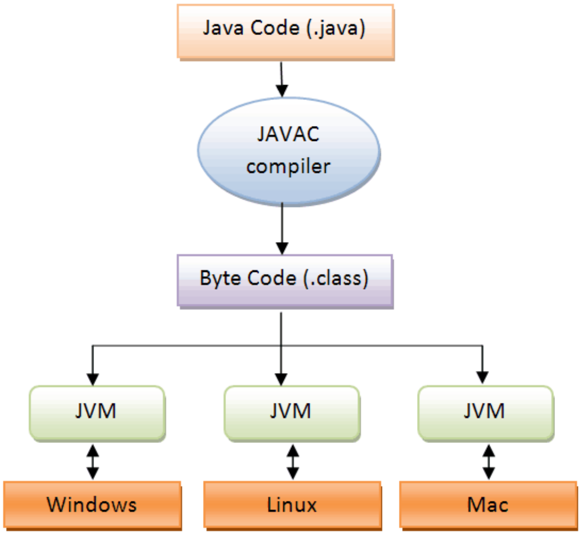
- http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2017/
- https://indeed.com

Java Background

- Designed by James Gosling 🍁
  - released by Sun Microsystems in 1995
  - Made open source under GNU GPL in 2007
  - Sun and Java acquired by Oracle in 2010
- Portable through virtualization
  - Requires Java Virtual Machine (JVM)
  - Is it compiled or interpreted?
    https://stackoverflow.com/questions/1326071/is-java-a-compiled-or-an-interpreted-programming-language
- Class-based, object-oriented design
  - C++ syntax, strongly typed
  - Manages memory
  - Extensive class libraries

Java Portability through Virtualization

- Java compiles to bytecode (.class file)
- Bytecode is executed by a Java Virtual Machine (JVM)
- Just-in-Time (JIT) bytecode compilation can give near-native performance.



Java Code (.java)

JAVAC compiler

Byte Code (.class)

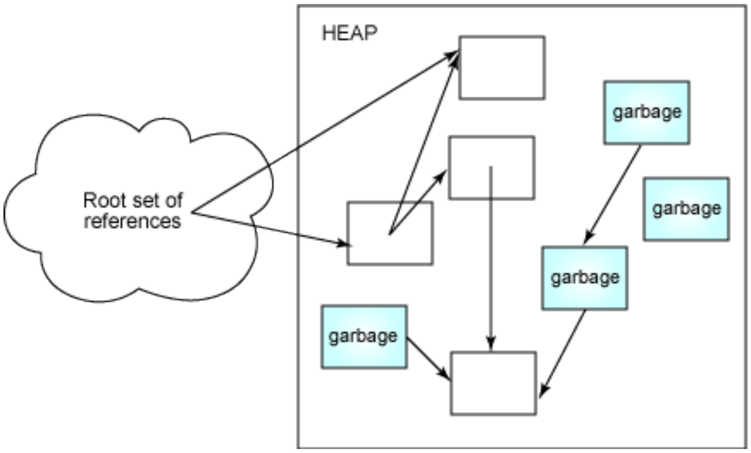
JVM JVM JVM

Windows Linux Mac

http://viralpatel.net/blogs/java-virtual-machine-an-inside-story/

Garbage Collection (GC)

- Garbage collection and frees up memory that’s not in use
- JVM *attempts* to do this without impacting performance



http://www.ibm.com/developerworks/library/j-jtp10283/

Why could this be bad?

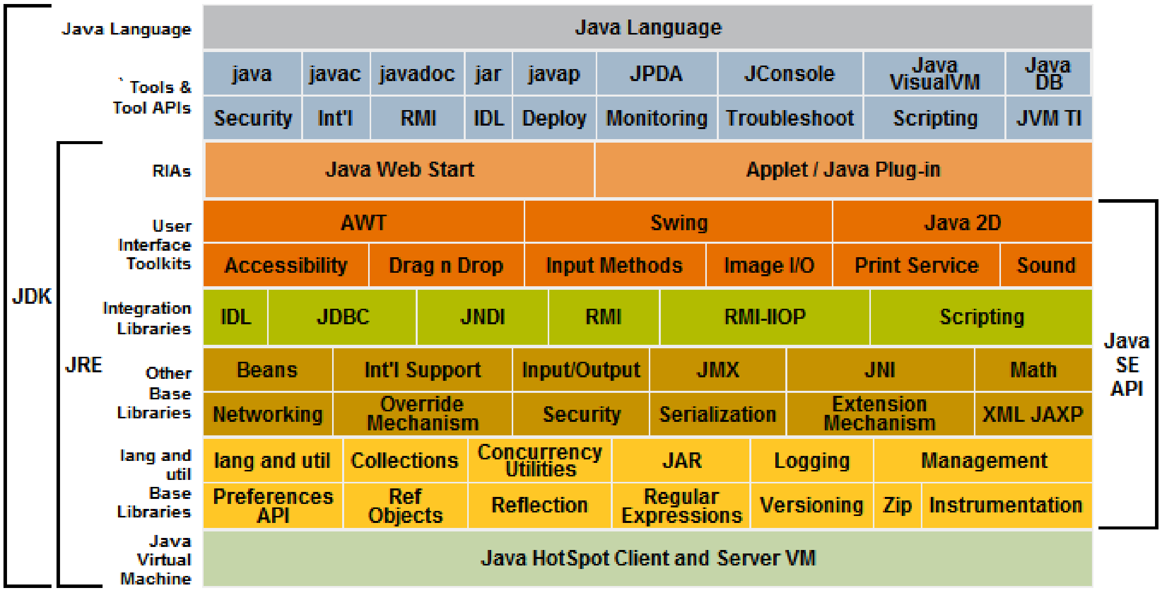
```
for (int i = 0; i < BIGNUM; i++) {
        BigFancyProcessor bfp = new BigFancyProcessor();
        send(bfp.process(data[i]));
}
```

(Almost) Everything is a Class

- **`Classes`** and objects are core constructs
- OO features: polymorphism, encapsulation, inheritance, …
- Static member variables and methods
- Resembles C++ on the surface, but not the same
  - No pointers, all references
  - No type ambiguity; classes resolved at runtime
  - No destructor (due to garbage collector)
  - No multiple inheritance (single only, but with class **`Interfaces`**)

Java Development Kit (JDK)

- cross platform and portable tools and libraries



JDK

JRE

Java SE API

Java Language: Java Language

` Tools & Tool APIs: java | javac | javadoc | jar | javap | JPDA | JConsole | Java VisualVM | Java DB

Security | Int'l | RMI | IDL | Deploy | Monitoring | Troubleshoot | Scripting | JVM TI

RIAs: Java Web Start | Applet / Java Plug-in

User Interface Toolkits: AWT | Swing | Java 2D

Accessibility | Drag n Drop | Input Methods | Image I/O | Print Service | Sound

Integration Libraries: IDL | JDBC | JNDI | RMI | RMI-IIOP | Scripting

Other Base Libraries: Beans | Int'l Support | Input/Output | JMX | JNI | Math

Networking | Override Mechanism | Security | Serialization | Extension Mechanism | XML JAXP

lang and util Base Libraries: lang and util | Collections | Concurrency Utilities | JAR | Logging | Management

Preferences API | Ref Objects | Reflection | Regular Expressions | Versioning | Zip | Instrumentation

Java Virtual Machine: Java HotSpot Client and Server VM

Java Class Library

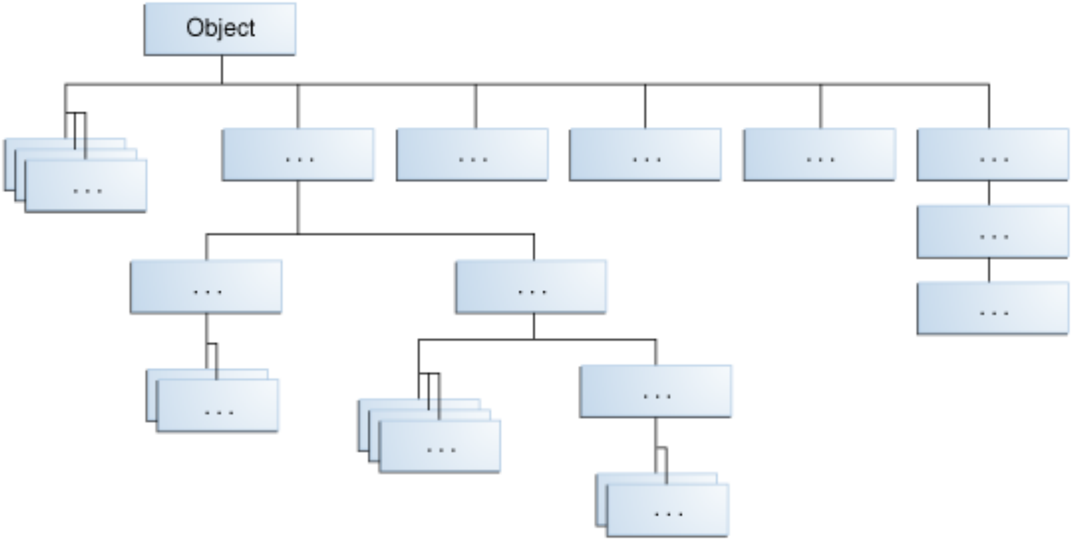
- Classes are grouped into ”packages”
- **`package`** keyword to assign source to a package
- Typically, a package is a subdirectory
  - e.g. “graphics” package is in subdirectory of the same name
- **`import`** keyword to include a class from a different package
  - This is how you include bundled Java libraries.

Common Classes/Packages

Package	Classes (Examples)	Description
java.awt	Color, Graphics, Graphics2D, event.	Contains all of the classes for creating user interfaces and for painting graphics and images.
javax.swing	JFrame, JButton, JList, JToolbar	Provides a set of "lightweight" (all-Java language) components that works the same on all platforms.
java.io	File, FileReader, FileWriter, InputStream	Provides for system input and output through data streams, serialization and the file system.
java.lang	Boolean, Integer, String, System, Thread, Math	Provides classes that are fundamental to the design of the Java programming language.
java.util	ArrayList, HashMap, Observable	Contains the collections framework, legacy collection classes, event model,…

Java Class Hierarchy

- All classes (implicitly) derive from Object class (in java.lang)
  has methods like clone(), toString(), finalize()
- Classes you write inherit these basic behaviours



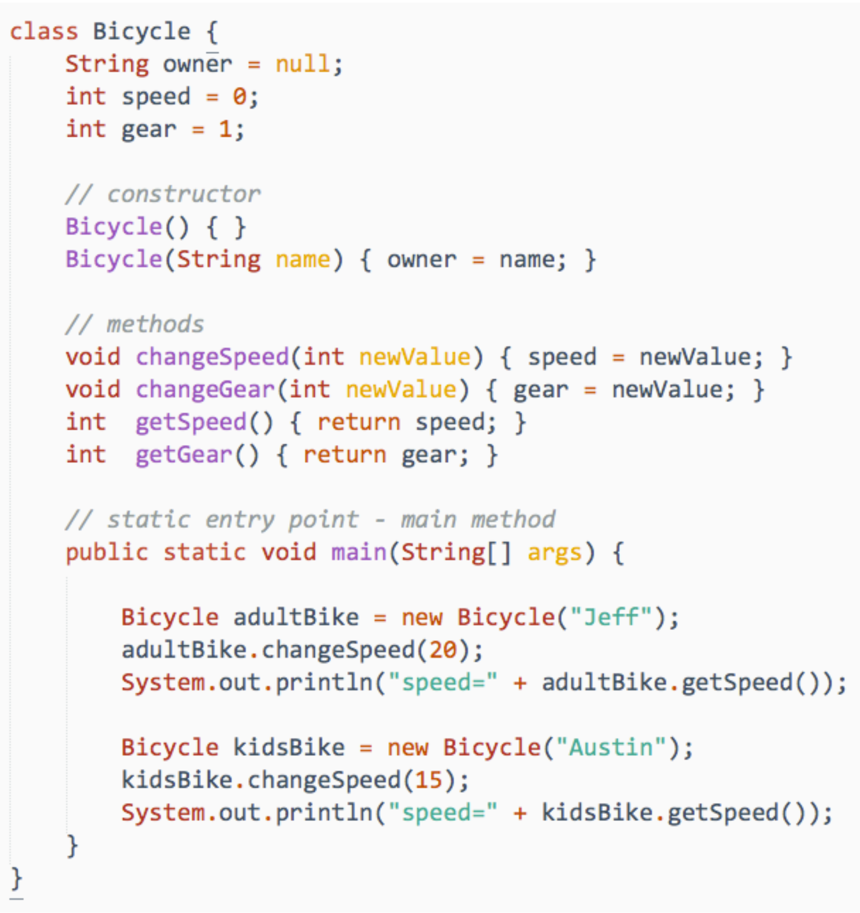
class

fields

constructor

methods

main

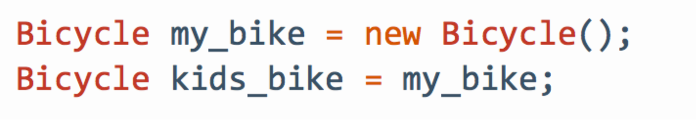


```
class Bicycle {
    String owner = null;
    int speed = 0;
    int gear = 1;

    // constructor
    Bicycle() { }
    Bicycle(String name) { owner = name; }

    // methods
    void changeSpeed(int newValue) { speed = newValue; }
    void changeGear(int newValue) { gear = newValue; }
    int  getSpeed() { return speed; }
    int  getGear() { return gear; }

    // static entry point - main method
    public static void main(String[] args) {

        Bicycle adultBike = new Bicycle("Jeff");
        adultBike.changeSpeed(20);
        System.out.println("speed=" + adultBike.getSpeed());

        Bicycle kidsBike = new Bicycle("Austin");
        kidsBike.changeSpeed(15);
        System.out.println("speed=" + kidsBike.getSpeed());
    }
}
```

Instantiating Objects

- Primitive types (**`int`**, **`float`**, etc.) are allocated on the stack
  - they are always *passed by value*
- Objects are allocated on the heap
  - you can think of them as always *passed by reference*
  - (in truth, object address is passed by value)
- There are no “pointer semantics” in Java
  - no **`*`**, no **`&`**, no **`out`**, no **`ref`**

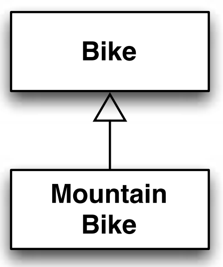
both refer to same memory on the heap



```
Bicycle my_bike = new Bicycle();
Bicycle kids_bike = my_bike;
```

Inheritance

- Inherit some methods or fields from a base class (“is a”)
- Very common in Java to inherit and override other classes
- Example:
  - “Mountain Bike” is-a “Bike”
  - Mountain bike inherits speed and gear fields
  - Mountain bike defines addition field for suspension type

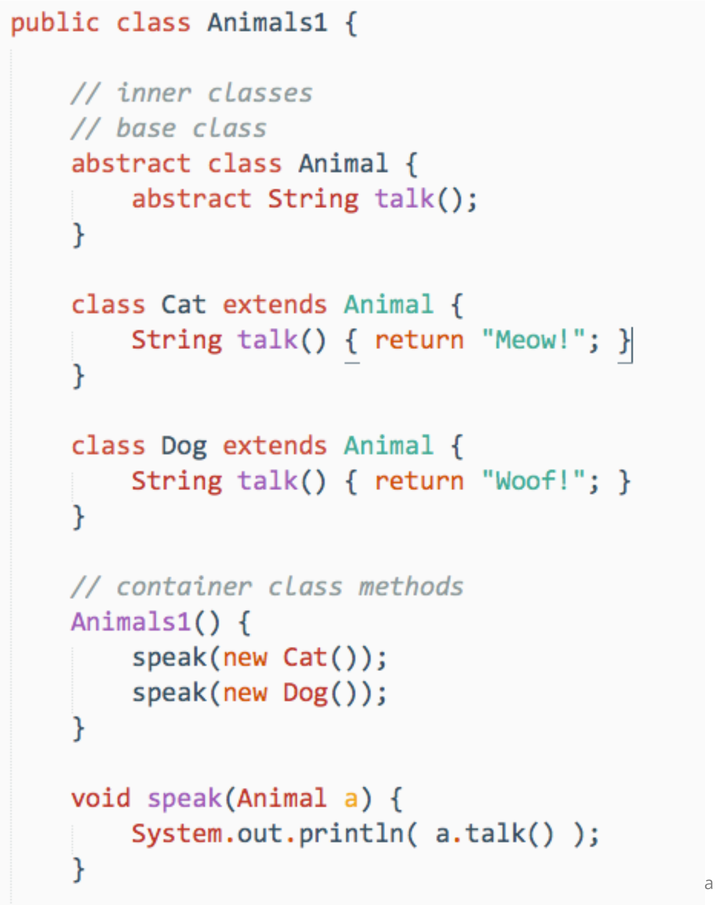


container class

abstract inner base class

inner inherited class

" Meow! "
" Woof! "



```
public class Animals1 {

    // inner classes
    // base class
    abstract class Animal {
        abstract String talk();
    }

    class Cat extends Animal {
        String talk() { return "Meow!"; }
    }

    class Dog extends Animal {
        String talk() { return "Woof!"; }
    }

    // container class methods
    Animals1() {
        speak(new Cat());
        speak(new Dog());
    }

    void speak(Animal a) {
        System.out.println( a.talk() );
    }
```

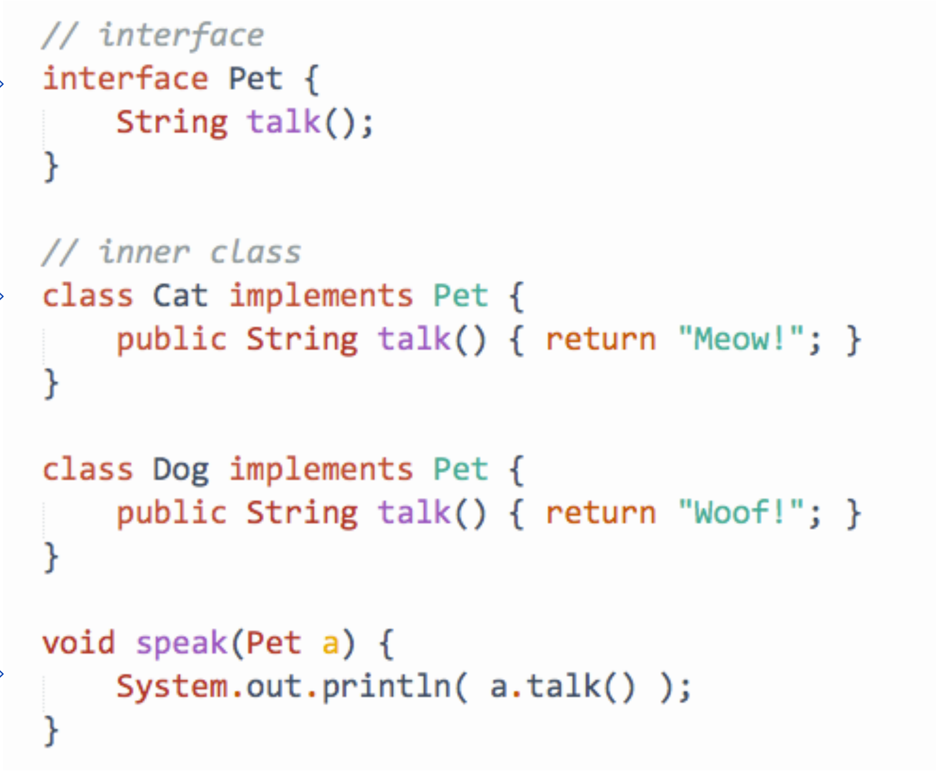
Interfaces

- An **`interface`** represents a set of methods a class must have
  - it's a "contract"
  - essentially, a pure abstract class
  - an interface can't be instantiated
- A class **`implements`** *all* methods in the interface
- A class can implement multiple interfaces
- Interfaces are used to enforce an API, not functionality

interface

implementations

The interface Pet is like a type

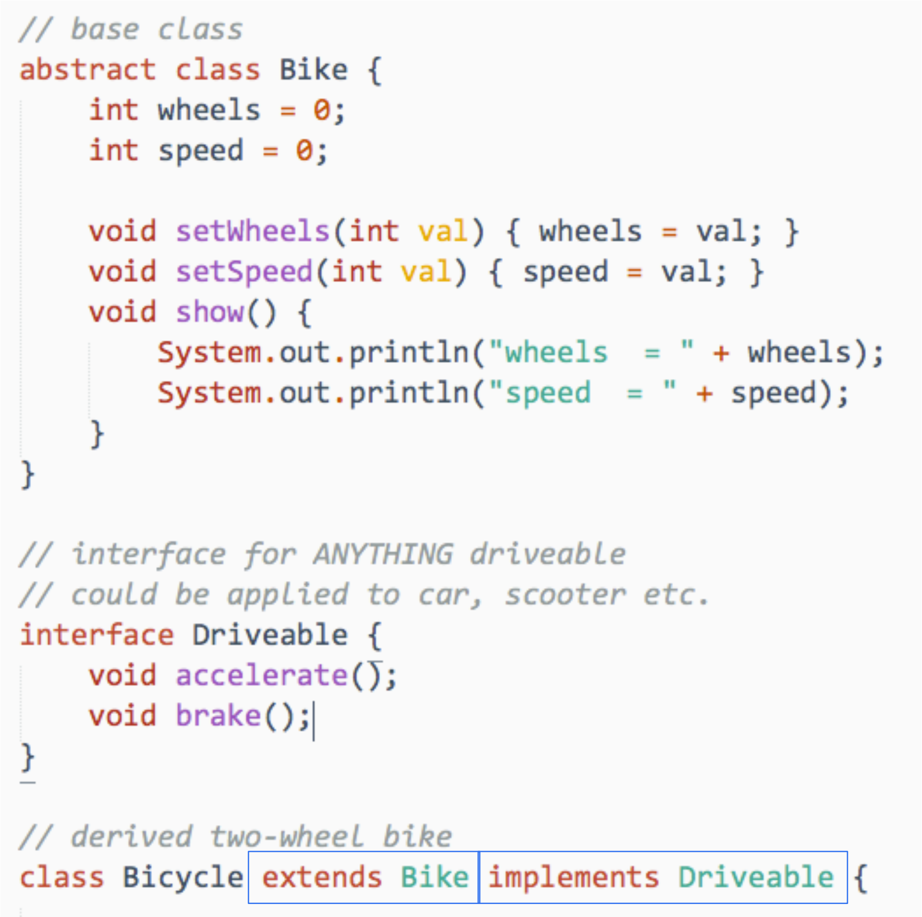


```java
// interface
interface Pet {
    String talk();
}

// inner class
class Cat implements Pet {
    public String talk() { return "Meow!"; }
}

class Dog implements Pet {
    public String talk() { return "Woof!"; }
}

void speak(Pet a) {
    System.out.println( a.talk() );
}
```

base class

interface

derived class



```java
// base class
abstract class Bike {
    int wheels = 0;
    int speed = 0;

    void setWheels(int val) { wheels = val; }
    void setSpeed(int val) { speed = val; }
    void show() {
        System.out.println("wheels  = " + wheels);
        System.out.println("speed  = " + speed);
    }
}

// interface for ANYTHING driveable
// could be applied to car, scooter etc.
interface Driveable {
    void accelerate();
    void brake();
}

// derived two-wheel bike
class Bicycle extends Bike implements Driveable {
```

Hello Java

```
import javax.swing.*;
import java.awt.Font;

public class Hello extends JFrame {

   public static void main(String args[]) {
      new Hello();
   }

   Hello() {
      JLabel l = new JLabel("Hello Java");
      l.setFont(new Font("Serif", Font.PLAIN, 24));
      add(l);
      setSize(200, 100);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
   }
}
```