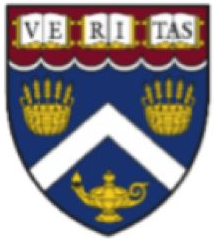
CSCI E-88A Introduction to Functional and Stream Processing for Big Data Systems

Harvard University Extension, Spring 2020
Marina Popova, Edward Sumitra



Lecture 2 - Introduction To Java Concepts

Agenda

- Admin info: Lab
- Java World - demystified
- Intro to Basic Java concepts (classes, methods, variables)
- Compilation and Execution
- Beyond Basics: Inheritance and Abstraction

Java: The Beginning

Java celebrated it 25-th year anniversary this year!

- developed originally by James Gosling and team at Sun Microsystems (acquired by Oracle later)
- first release: 1995
- since Oracle acquisition - releases are not totally free
- last free release: JDK1.8 (that's why everybody still using it ...)
- there are other open source implementations that can be used:
  - OpenJDK - open source code of Java Se (Standard Edition)
  - AdoptOpenJDK - binary distributions of OpenJDK
- we are going to use OpenJDK 11 version in this class

For more info: https://developer.okta.com/blog/2019/01/16/which-java-sdk

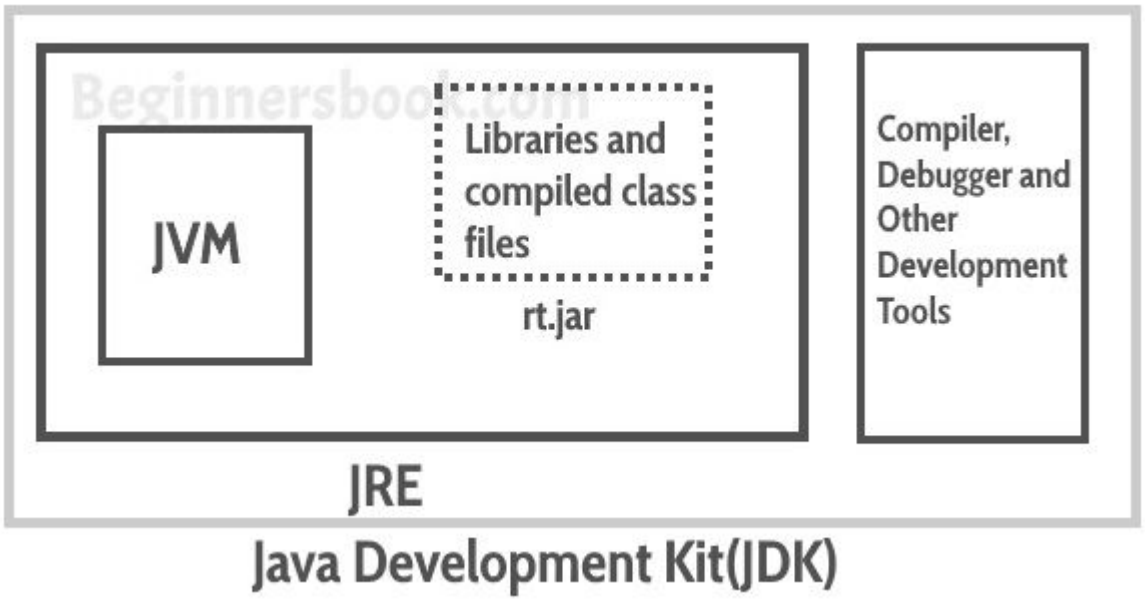
Java Architecture

JDK (Java Development Kit) - dev kit for building apps using Java

- includes dev tools and JRE
- required for development

Dev Tools:

- javac - compiler
- java - launcher/interpreter
- jar - archiver
- many more ...



JRE (Java Runtime Environment): a set of tools to launch compiled Java programs; includes:

- Java core libraries (utils, math, lang, ...)
- compiled classes
- runtime libraries (rt.jar)
- JVM (Java Virtual Machine)

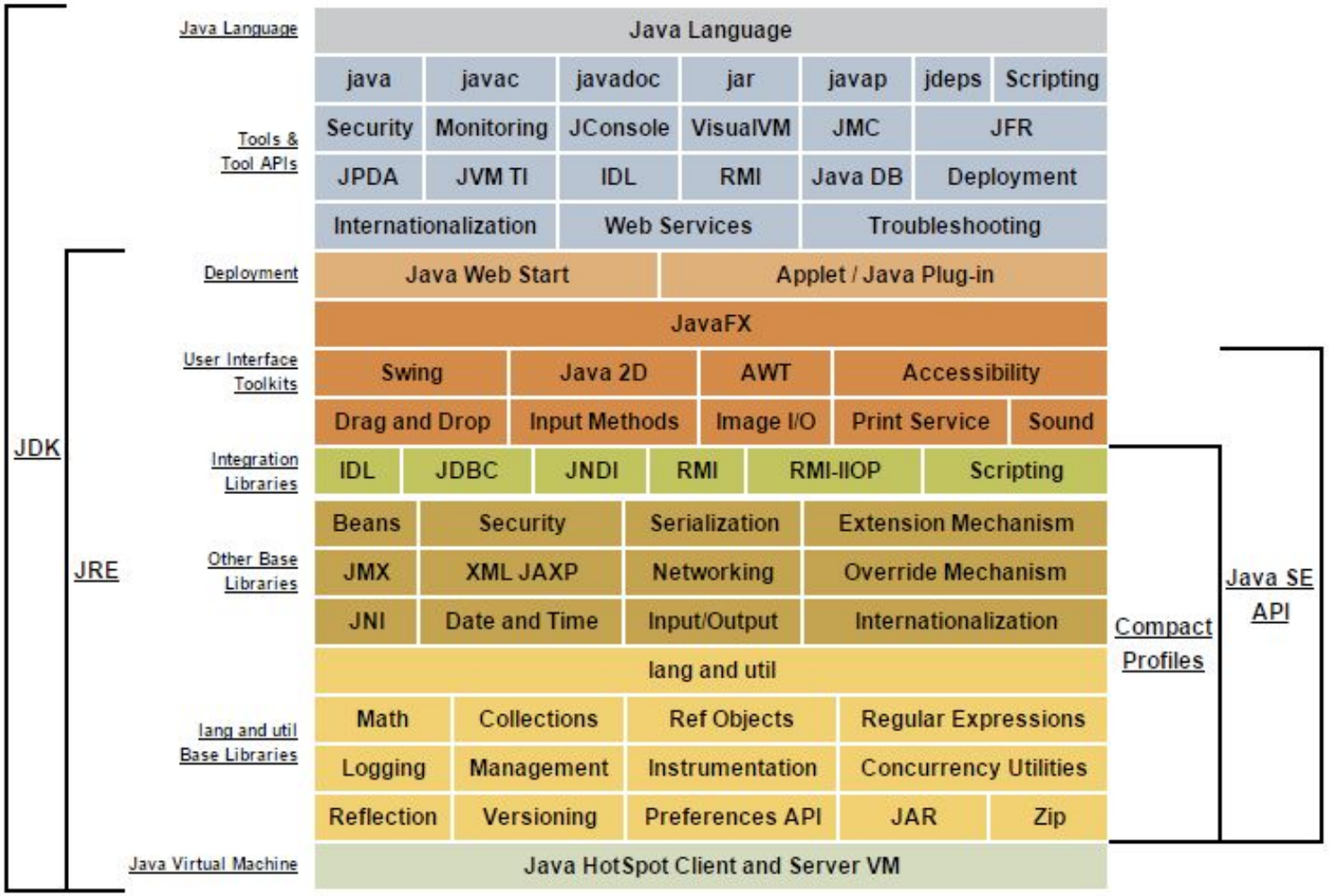
JVM (Java Virtual Machine): a program responsible for execution of the compiled Java code (bytecode); contains:

- classloader system
- execution engine
- runtime data area

Refs: https://www.geeksforgeeks.org/differences-jdk-jre-jvm/
https://beginnersbook.com/2013/05/jvm/

Java Architecture: Overview

Description of Java Conceptual Diagram

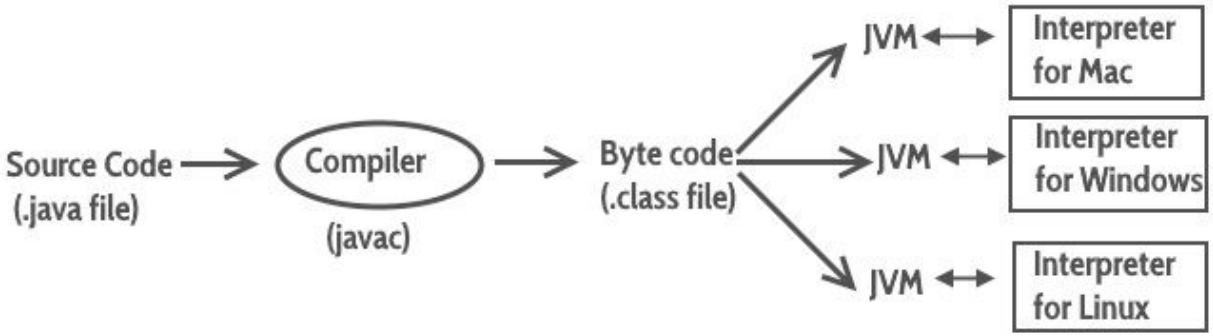


Refs: https://www.programsbuzz.com/question/what-difference-between-jdk-jre-and-jvm

Java Architecture: JVM

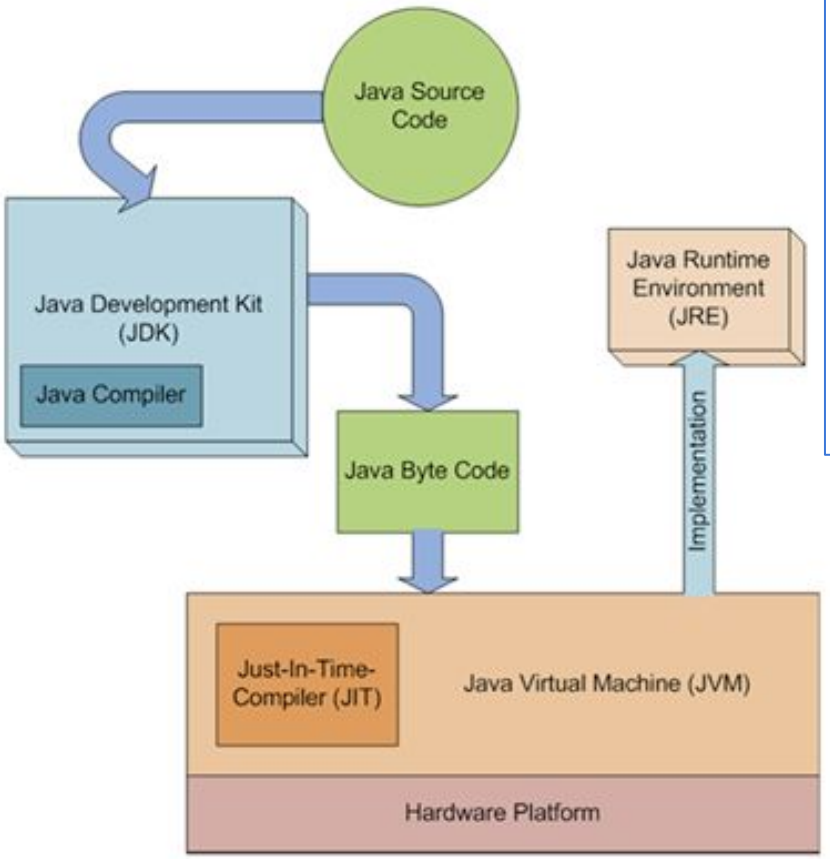
JVM is what makes Java platform (OS/hardware) independant and makes the "write once, run everywhere" promise fulfilled!

Life cycle of a Java app: source code to byte code to be interpreted by the JRE and gets converted to the platform specific executable ones



Refs: https://www.geeksforgeeks.org/differences-jdk-jre-jvm/
https://beginnersbook.com/2013/05/jvm/

Java Architecture - Deeper Dive



- **Java Virtual Machine (JVM)** is an abstract computing machine.
- **Java Runtime Environment (JRE)** is an implementation of the JVM.
- **Java Development Kit (JDK)** contains JRE along with various development tools like Java libraries, Java source compilers, Java debuggers, bundling and deployment tools.
- **Just In Time compiler (JIT)** is runs after the program has started executing, on the fly. It has access to runtime information and makes optimizations of the code for better performance.

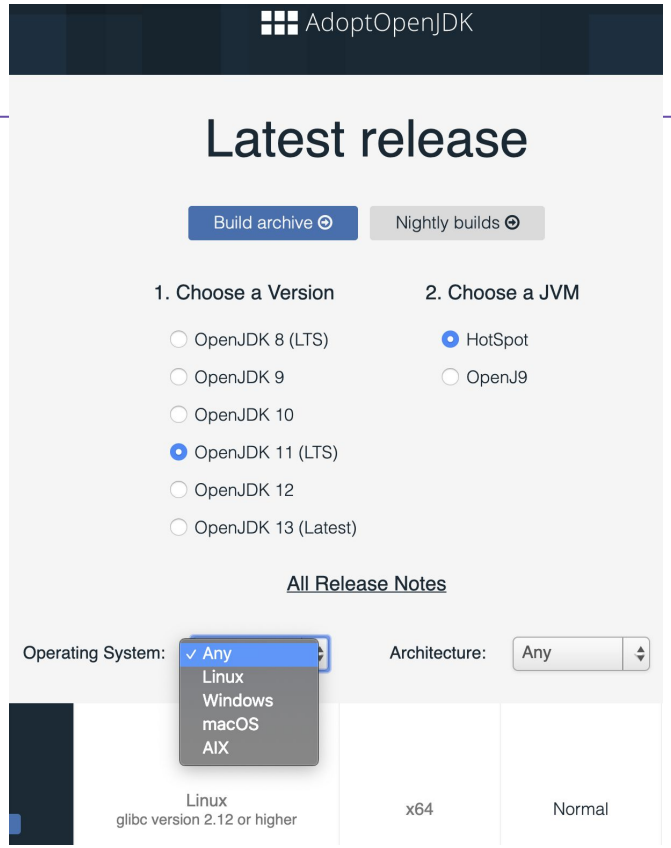
JVM becomes an instance of JRE at runtime of a Java program

... there are multiple implementations of JVM, available as vendor supplied JREs, for different hardware platforms ...

Ref: https://javapapers.com/core-java/differentiate-jvm-jre-jdk-jit/

Java Architecture: JVM

You pick specific implementation of JRE/JVM for a target OS/Architecture:



@Marina Popova, Edward Sumitra

Java Applications: Basics

How do you create Java applications? By writing "code"

- code in Java is stored in files with .java extension:
  - the name of the file is important - it has to be the same as the name of "class" the code is written for (more on this later)
- Java is OO language
- in procedural languages, like C, you usually have data structures and instructions to operate on them
- in OO language, data and related functionality are usually co-located in "objects"
- "an **object** is a self-contained entity that contains attributes and behavior, and nothing more"

Ref: https://developer.ibm.com/tutorials/j-introtojava1/

Java Applications: Basics

- Objects contain both attributes and behavior
- attributes is what is referred to as "state" of an object
  - attributes can be of different types (String, Integer, long, other class types)
  - attributes can have different visibility (private, public, ...)
- behavior is defined in "methods"
  - methods can also have different visibility
  - methods can take parameters
  - methods can return one value (of primitive or complex type)

```java
package cscie88a.basics2;

public class Cat {

    private String name;
    private String eyeColor;
    private String bodyColor;

    public Cat() {}

    public Cat(String name, String eyeColor, String bodyColor) {
        super();
        this.name = name;
        this.eyeColor = eyeColor;
        this.bodyColor = bodyColor;
    }

    public String saySomething(String somethingToSay){
        String whatISay = "I don't care what you asked me to say - I say MEOW only";
        return whatISay;
    }
```

Java Applications: Basics

How do you design Objects?

How do you decide which attributes (state) and behavior (methods) should belong to which object?

It's an Art and an acquired skill - that you'll be getting better and better at the more you do it

Lets look at our Cat aggregation example again....

Cat Aggregator

```java
package cscie88a.basics2;

import java.util.ArrayList;

public class CatAggregator {

    public CatAggregator() {
    }

    public long countCatsByColorImperative(
        Collection<Cat> allCats, String bodyColorToMatch, String eyeColorToMatch) {

        Long numOfCats = 0l;
        for (Cat cat: allCats) {
            if (cat.getBodyColor().equalsIgnoreCase(bodyColorToMatch) &&
                cat.getEyeColor().equalsIgnoreCase(eyeColorToMatch)) {
                numOfCats++;
            }
        }
        System.out.println("Imperative: Found " + numOfCats + " " +
            bodyColorToMatch + " cats with " + eyeColorToMatch + " eyes");
        return numOfCats;
    }
```

```java
package cscie88a.basics2;

public class Cat {

    private String name;
    private String eyeColor;
    private String bodyColor;

    public Cat() {}

    public Cat(String name, String eyeColor, String bodyColor) {
        super();
        this.name = name;
        this.eyeColor = eyeColor;
        this.bodyColor = bodyColor;
    }

    public String saySomething(String somethingToSay){
        String whatISay = "I don't care what you asked me to say - I say MEOW only";
        return whatISay;
    }
```

- where do 'name', 'eyeColor', 'bodyColor' belong to?
- where does saySomething() method belong?
- where do countCatsBy...() methods belong to?
    - who cares about them? Cats? or CatAggregators?

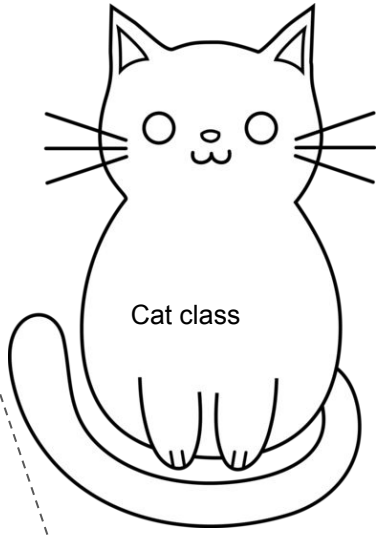
Classes vs Objects vs Instances

So far we were using the terms "class" and "object" interchangeably... but they are not one and the same thing ...

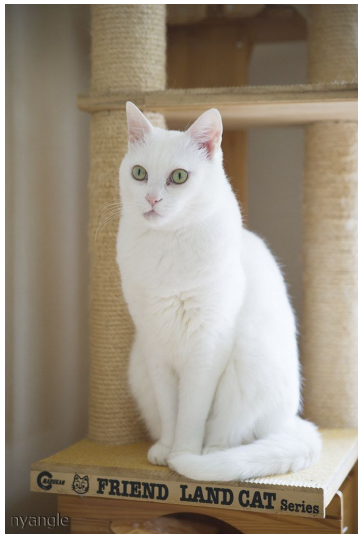
Class is a template, a blueprint, a definition, for an object to be created from; is declared in the .java file

Object - a concrete instance of a Class; crated via new()

Object == Instance



Cat instance 1



Cat instance 2

Cat instance 3

Image source:
http://clipart-library.com/clipart/pT5raM57c.htm

Classes vs Objects vs Instances

No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from which objects are created.
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g. class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.
6)	Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

Ref: https://www.javatpoint.com/difference-between-object-and-class

Java Basics: Class Structure

How do we define a class?

There are some syntax rules:

- **packages**: identify namespaces for classes
- **import statements**: allow use of other libraries/packages/classes
- **class ClassName**: identifies the class
- ClassName(...): constructor[s]
- variables
- methods

```
package packageName;
import ClassNameToImport;
accessSpecifier class ClassName {
  accessSpecifier dataType variableName [= initialValue];
  accessSpecifier ClassName([argumentList]) {
    constructorStatement(s)
  }
  accessSpecifier returnType methodName ([argumentList]) {
    methodStatement(s)
  }
  // This is a comment
  /* This is a comment too */
  /* This is a
     multiline
     comment */
```

Ref: https://developer.ibm.com/tutorials/j-introtojava1/

Java Basics: Class Structure - Example

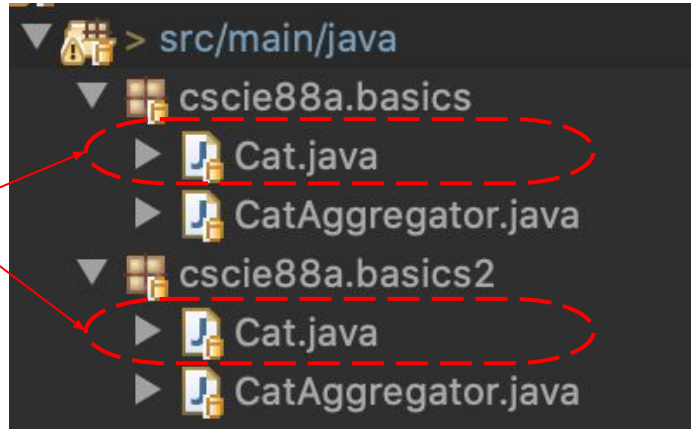
```
package cscie88a.basics2;

import java.util.ArrayList;

public class CatAggregator {

    public CatAggregator() {
    }

    public long countCatsByColorImperative(
            Collection<Cat> allCats, String bodyColorToMatch, String eyeColorToMatch) {

        long numOfCats = 0l;
        for (Cat cat: allCats) {
            if (cat.getBodyColor().equalsIgnoreCase(bodyColorToMatch) &&
                cat.getEyeColor().equalsIgnoreCase(eyeColorToMatch)) {
                numOfCats++;
            }
        }
        System.out.println("Imperative: Found " + numOfCats + " " +
            bodyColorToMatch + " cats with " + eyeColorToMatch + " eyes");
        return numOfCats;
    }
```

Java Basics: More On Packages ...

- packages identify **namespaces** for classes
- they use "." to separate multiple sub-packages
- fully qualified name of a Java class included both package and class name
- they correspond one-to-one to the file directory structure the .java files is located in
- once compiled - the generated byte code (.class file) will be created in the EXACT SAME directory structure

```
pmarina@pmarina-mac-0 localsetup % tree
.
├── myclasses
│   └── cscie88a
│       └── basics
│           ├── Cat.class
│           └── CatAggregator.class
└── src
    └── cscie88a
        └── basics
            ├── Cat.java
            └── CatAggregator.java

6 directories, 4 files
pmarina@pmarina-mac-0 localsetup %
```

not the same !!



Java Core Concepts: Static vs Instance Methods

- Objects can reference and access other objects via methods, and access their state either directly or via getter/setter methods
- There are two types of methods and variables
  - instance
  - static

Instance methods	Static methods
can be called on an instance of a class only, after it is created	can be called on the Class itself
can use Instance's state (variables)	can use class variables only
each instance of a class has their own copy of the method	there is only one copy of the method - associated with the class itself
can be overwritten	cannot be overwritten

static variables have similar characteristics to static methods

Ref: https://developer.ibm.com/tutorials/j-introtojava1/

@Marina Popova, Edward Sumitra

Java Core Concepts: Static vs Instance - Example

```java
public class Cat {

    public static String whatISay = "I don't care what you asked me to say - I say MEOW only";

    private String name;
    private String eyeColor;
    private String bodyColor;

    public Cat() {}

    public Cat(String name, String eyeColor, String bodyColor) {
        super();
        this.name = name;
        this.eyeColor = eyeColor;
        this.bodyColor = bodyColor;
    }

    public static String saySomething(String somethingToSay){
        return whatISay;
    }

    public static void main(String[] args) throws Exception {
        Cat demon = new Cat("Demon", "green", "black");
        Cat sneaky = new Cat("Sneaky", "blue", "gray");
        String somethingToSay = "Hello!";
        System.out.println("Demon says: " + demon.saySomething(somethingToSay));
        System.out.println("Sneaky says: " + sneaky.saySomething(somethingToSay));

        System.out.println("All cats say: " + Cat.saySomething(somethingToSay));
    }
```

the magic main() method

In order to be able to run an application - we have to have main() method defined in a class that is the starting point of your application. The main method has the following signature:

```
public static void main(String[] args)
```

The signature is very important:

- **public**: allows other classes/applications to call this method - JVM in this case
- **static**: does not require creation of an instance of the class in order to execute the method - so that JVM can execute the method before it can create an instance
- **void**: no return values are expected from the main method - why?

as soon as all instructions in the main() method are executed - the application exits - and the JVM stops !

Ref: https://developer.ibm.com/tutorials/j-introtojava1/

the magic main() method

Lets see how we can add this method to our CatAggregator to make it executable

- and make it run forever
- lets compile and execute it ....

```
public class CatAggregator {

    public CatAggregator() {
    }

    public long countCatsByColorImperative(

    public long countCatsByColorFunctional(Collection<Cat> allCats, String body

    public static void main(String[] args) throws Exception {
        Collection<Cat> testCats = new ArrayList<Cat>();
        testCats.add(new Cat("Demon", "green", "black"));
        testCats.add(new Cat("Sneaky", "blue", "gray"));
        testCats.add(new Cat("Angel", "blue", "white"));
        testCats.add(new Cat("Pirate", "green", "black"));
        testCats.add(new Cat("Scruffy", "golden", "black"));
        testCats.add(new Cat("Princess", "green", "white"));

        CatAggregator catAggregator = new CatAggregator();
        catAggregator.countCatsByColorFunctional(testCats, "black", "green");
        catAggregator.countCatsByColorImperative(testCats, "black", "green");

        // lets make this run forever ...
        while (true) {
            System.out.println("I'm sleeping ....");
            Thread.sleep(5000l);
        }
    }
}
```

Ref: https://developer.ibm.com/tutorials/j-introtojava1/

Java Applications: Lifecycle walkthrough again

- we create definitions/code of classes in the **.java files**
- we run a Java compiler, "**javac**", to produce bytecode from them - in the form of **.class files**
- the bytecode is loaded and interpreted by the JVM - when we start an instance of it via '**java**' launcher

```
pmarina@pmarina-mac-0 localsetup % ls -la src/cscie88a/basics
total 16
drwxr-xr-x  4 pmarina  staff   128 Feb  2 14:18 .
drwxr-xr-x  4 pmarina  staff   128 Feb  2 13:58 ..
-rw-r--r--  1 pmarina  staff  1749 Jan 26 13:56 Cat.java
-rw-------  1 pmarina  staff  1928 Feb  2 14:18 CatAggregator.java
pmarina@pmarina-mac-0 localsetup % java --version
java 13.0.1 2019-10-15
Java(TM) SE Runtime Environment (build 13.0.1+9)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)
pmarina@pmarina-mac-0 localsetup % javac src/cscie88a/basics/*.java -d myclasses
pmarina@pmarina-mac-0 localsetup % ls -la myclasses/cscie88a/basics
total 16
drwxr-xr-x  4 pmarina  staff   128 Feb  2 14:11 .
drwxr-xr-x  3 pmarina  staff    96 Feb  2 14:11 ..
-rw-r--r--  1 pmarina  staff  2040 Feb  2 14:31 Cat.class
-rw-r--r--  1 pmarina  staff  3977 Feb  2 14:31 CatAggregator.class
pmarina@pmarina-mac-0 localsetup % java -classpath "./myclasses" cscie88a.basics.CatAggregator
Functional: Found 2 black cats with green eyes
Imperative: Found 2 black cats with green eyes
I'm sleeping ....
^C%
pmarina@pmarina-mac-0 localsetup %
```

Java Applications: The Dreaded Classpath

What is a Classpath in Java? And what is a Classloader?

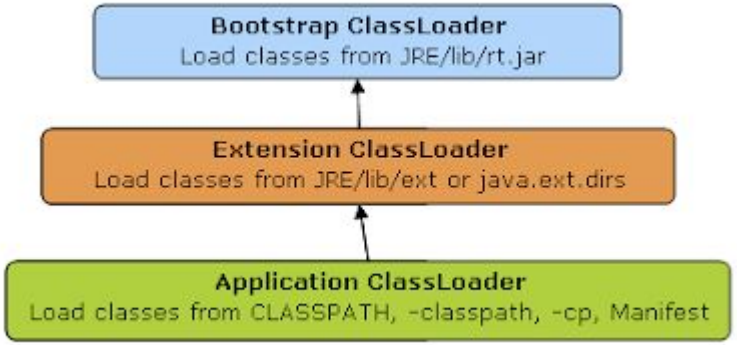
- when we say "the bytecode is loaded and interpreted by the JVM" - the "loading" part is a deep topic in itself ...
- compiled classes, the bytecode, are loaded into the JVM by a special class, Classloader
- There is not one, but a few hierarchical Classloaders used by JVM - to load different groups of classes from different locations
- The Application Classloader is responsible for loading classes from a location (file system directories) specified by the CLASSPATH environment variable
- this CLASSPATH can also be specified on a command line when invoking the 'java' launcher - by using the "-classpath" or "-cp" flag

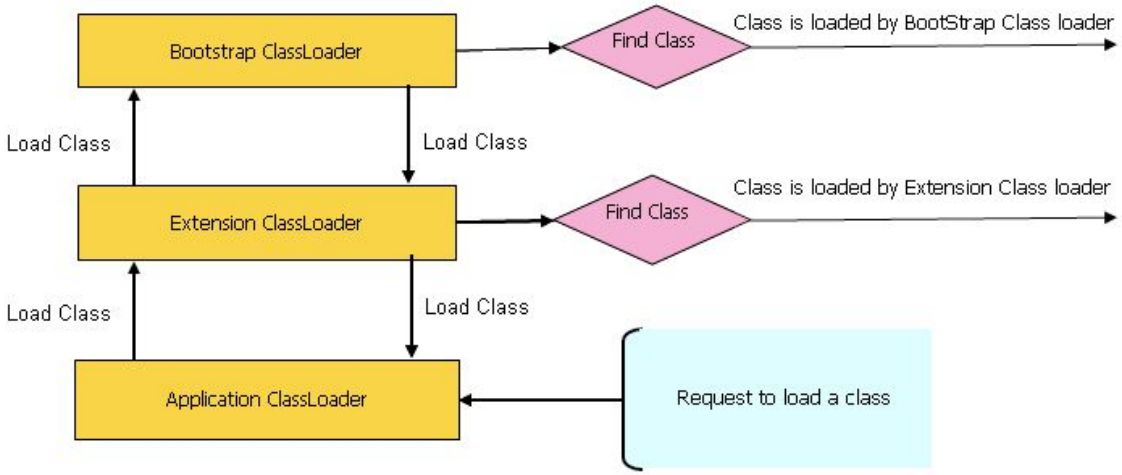
Refs: https://www.baeldung.com/java-classloaders
https://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html

@Marina Popova, Edward Sumitra

More on Classloaders

- classloaders delegate requests to load a class to a parent classloader first
- and only if the parent classloader cannot load the class - the child classloader will attempt to do so
- classes are loaded "on demand"

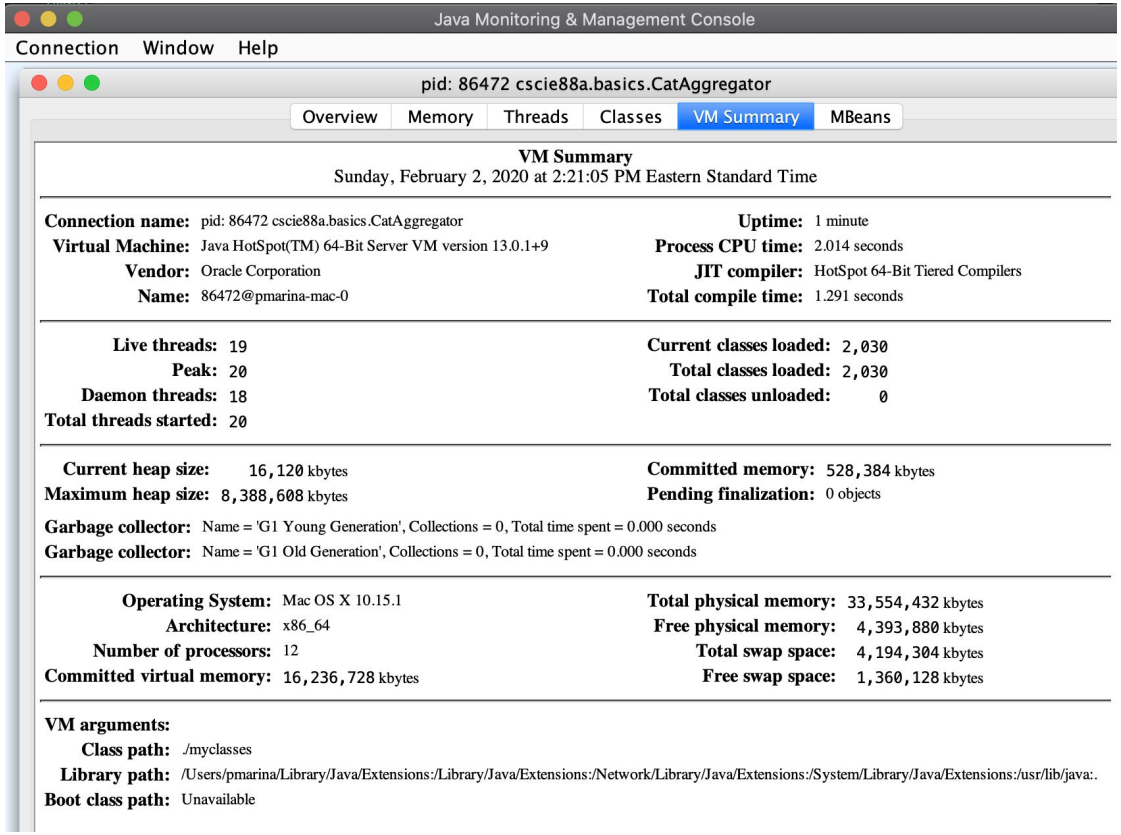




Refs: https://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html

Java Applications: Classpath and JConsole

- '**jconsole**' utility comes as part of the JDK distribution
- you can attach it to a running JVM process - local or remote



Core OO Concepts

- Objects can be composed into hierarchies with parent-child relationships
- a parent class is what is called a "super" class, and child classes "extend" the parent class

these concepts play important role in enabling the three main principles of the OOP:

Encapsulation: controlling the visibility/access to the state and behavior of the objects from other objects

Inheritance: ability of more specialized classes (children) to re-use and/or extend the functionality of less specialized classes (parents)

Polymorphism: ability to change behavior or type in the same object hierarchy
Achieved by method overloading and method overriding

For more info on polymorphism: https://www.geeksforgeeks.org/polymorphism-in-java/

Ref: https://developer.ibm.com/tutorials/j-introtojava1/

Java Core Concepts: Inheritance - Example

Lets evolve our Cats example a bit:

add Animal - parent class
new common method: takeMedicine()
Cat and Dog - children classes

```java
package cscie88a.basics3;

public class Animal {

    protected String name;
    protected String eyeColor;
    protected String bodyColor;

    public Animal() {
        super();
    }

    public Animal(String name, String eyeColor, String bodyColor) {
        super();
        this.name = name;
        this.eyeColor = eyeColor;
        this.bodyColor = bodyColor;
    }

    public boolean takeMedicine(boolean withTreat) {
        if (withTreat) {
            return true;
        } else {
            return false;
        }
    }
```

```java
package cscie88a.basics3;

public class Cat extends Animal {

    public static String whatISay = "I don't care what you asked me to

    public Cat() {}

    public Cat(String name, String eyeColor, String bodyColor) {
        super(name, eyeColor, bodyColor);
    }

    public static String saySomething(String somethingToSay){
        return whatISay;
    }

    // this method is overwritten !
    public boolean takeMedicine(boolean withTreat) {
        System.out.println("you won't trick me - I'm not taking it!");
        return false;
    }
```

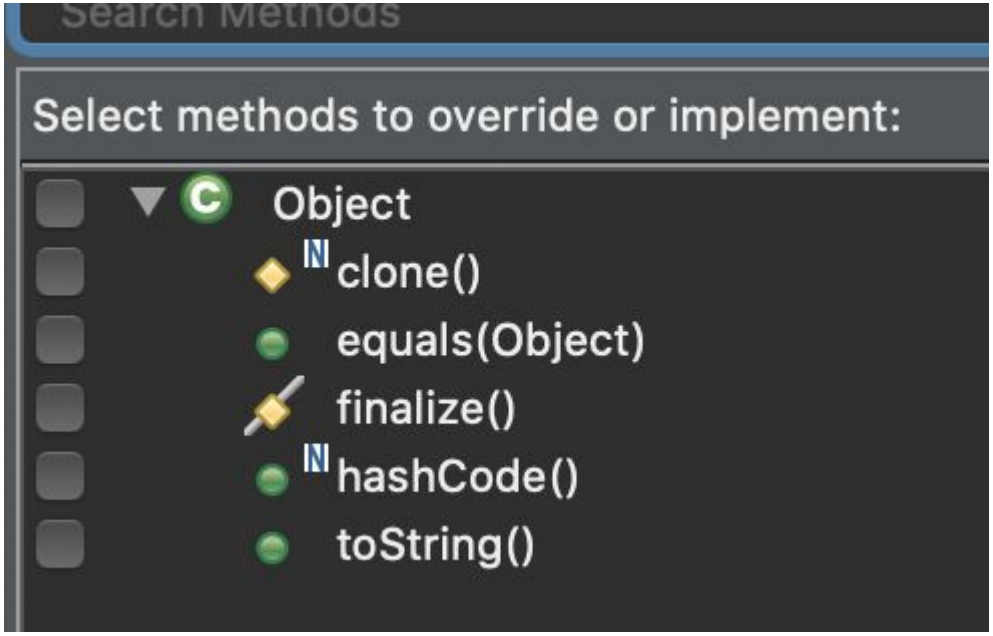
```java
package cscie88a.basics3;

public class Dog extends Animal {

    public Dog() {
    }

    public Dog(String name, String eyeColor, String bodyColor) {
        super(name, eyeColor, bodyColor);
    }

}
```

@Marina Popova, Edward Sumitra

Java Core Concepts: Inheritance

All Java objects extend a special core Java object call Object

Thus, all objects inherit behavior from the Object:



Java Core Concepts: encapsulation

Methods and variables in Java can have different **levels of access.**

Access level modifiers determine whether other classes can use a particular field or invoke a particular method

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Ref: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

@Marina Popova, Edward Sumitra

Java: Beyond Basics: Abstract classes

Abstract Classes are almost like "incomplete" regular classes :)

- must have at least one empty (unimplemented) method
- declared with "abstract" keyword
- cannot be instantiated directly
- children classes have to implement all abstract methods, or declare themselves as abstract as well

@Marina Popova, Edward Sumitra

Java: Beyond Basics: Abstract classes - Example

Lets introduce AbstractAnimal class, with one abstract method:

void sayHiToHuman(String humanName)

```java
package cscie88a.basics4;

public class Dog extends AbstractAnimal {

    public Dog() {

    public Dog(String name, String eyeColor, String bodyColor) {

    @Override
    public void sayHiToHuman(String humanName) {
        System.out.println("Hi, " + humanName + "!!! I LOVE you!");
    }
```

```java
public abstract class AbstractAnimal {

    protected String name;
    protected String eyeColor;
    protected String bodyColor;

    public AbstractAnimal() {

    public AbstractAnimal(String name, String eyeColor, String bodyColor) {

    public abstract void sayHiToHuman(String humanName);

    public boolean takeMedicine(boolean withTreat) {
        if (withTreat) {
            return true;
        } else {
            return false;
        }
    }
```

```java
package cscie88a.basics4;

public class Cat extends AbstractAnimal {

    public static String whatISay = "I don't care what you asked me
    public static String humanGreeting = "Go away ...";

    public Cat() {}

    public Cat(String name, String eyeColor, String bodyColor) {

    public static String saySomething(String somethingToSay){
        return whatISay;
    }

    @Override
    public void sayHiToHuman(String humanName) {
        System.out.println(humanGreeting);
    }
```

Java: Beyond Basics: Enums

- Enums are a special type of Java classes - used to define collections of constants
- defined with the 'enum' keyword
- enums can also contain methods and variables
- can be used as method parameters, return types, in different syntax constructs
- for more details: http://tutorials.jenkov.com/java/enums.html

```
package cscie88a.basics4;

public enum ActionResult {
    SUCCESS,
    FAILURE
}
```

```
@Override
public ActionResult playWithMe(AbstractAnimal aFriend) {
    // I'll play only if I'm in the mood ...
    if (isFriendly)
        return ActionResult.SUCCESS;
    else
        return ActionResult.FAILURE;
}
```

```
@Test
public void testPlayWithMe() {
    // unfriendly cat will not play with anyone
    sneaky.setFriendly(false);
    ActionResult result = sneaky.playWithMe(bolt);
    assertEquals(ActionResult.FAILURE, result);

    // a friendly cat will play with others
    sneaky.setFriendly(true);
    result = sneaky.playWithMe(bolt);
    assertEquals(ActionResult.SUCCESS, result);

    // dog will play with anybody, always
    result = bolt.playWithMe(sneaky);
    assertEquals(ActionResult.SUCCESS, result);
}
```

```
@Override
public ActionResult playWithMe(AbstractAnimal aFriend) {
    // I am happy to play with anyone!
    return ActionResult.SUCCESS;
}
```

Java: Beyond Basics: Interfaces

- Interfaces are used to define a set of related behavior
- can be thought of as a "blueprint" of a behavior for classes
- they usually contain abstract methods with no implementation
- can have final and static fields
- classes have to implement interfaces using the "**implements**" keyword
- a class can implement more than one interfaces
- interfaces can also have default and static method - more later

Interfaces allow us to abstract definition of behavior from its implementation

Refs: https://www.geeksforgeeks.org/interfaces-in-java/

@Marina Popova, Edward Sumitra

Java: Beyond Basics: Interfaces - Example

Lets add an interface, ITrainable, with one method: *doTrick()* and make both Cat and Dog implement it

```java
public interface ITrainable {

    public ActionResult doTrick(String trickName);

}
```

```java
public class Cat extends AbstractAnimal implements ITrainable{

    public static String whatISay = "I don't care what you asked me t
    public static String humanGreeting = "Go away ...";
    private boolean isFriendly = false;

    public Cat() {}

    public Cat(String name, String eyeColor, String bodyColor) {

    @Override
    public ActionResult doTrick(String trickName) {
        System.out.println(name + " says: cats do NOT do tricks ");
        return ActionResult.FAILURE;
    }
```

```java
public class Dog extends AbstractAnimal implements ITrainable{

    public Dog() {

    public Dog(String name, String eyeColor, String bodyColor) {

    public void sayHiToHuman(String humanName) {

    public ActionResult playWithMe(AbstractAnimal aFriend) {

    @Override
    public ActionResult doTrick(String trickName) {
        System.out.println(name + " says: I LOVE doing tricks! I'm doing " + trickName + " now!");
        return ActionResult.SUCCESS;
    }
```

Java: Beyond Basics: Interfaces - default methods

Since Java 8 - interfaces can have default methods

- main purpose - make interfaces extensible
  - ability to add new methods to interfaces without requiring ALL implementing classes to implement them
- default methods can only declare behavior - they cannot use/access implementing instances' state (non-final and non-static variables)
- implementing classes can override the default methods with their own implementation if desired

- interfaces can also have static methods - similar to regular classes - which cannot access any state

Ref: http://chaseyourjava.blogspot.com/2017/12/default-methods-in-java-8.html

Interface default methods - example

```
public interface ITrainable {

    public ActionResult doTrick(String trickName);

    default public ActionResult doTrickForTreat(String trickName, String treatName) {
        System.out.println("I love the " + treatName +
            " and will happily do the trick [" + trickName + "] !!");
        return ActionResult.SUCCESS;
    }
}
```

Cat, of course, would have to override the method:

```
@Override
public ActionResult doTrickForTreat(String trickName, String treatName) {
    System.out.println(name + " says: cats do NOT do tricks ");
    return ActionResult.FAILURE;
}
```

Java: Beyond Basics: Interfaces vs Abstract classes

- abstract classes can define constructors
- non-abstract methods in abstract classes can use instance fields

Ref: https://www.topjavatutorial.com/java/interface-default-methods-vs-abstract-class-java-8/

From Oracle docs:

Consider using abstract classes if any of these statements apply to your situation:

- You want to share code among several closely related classes.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Consider using interfaces if any of these statements apply to your situation:

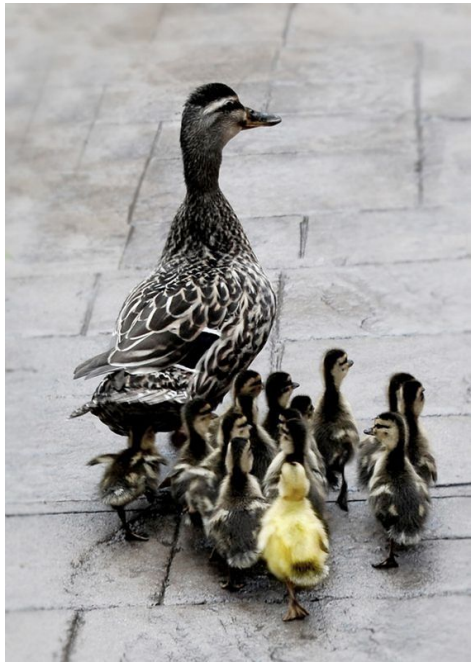
- You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.

Java: Beyond Basics: Anonymous classes

- Anonymous classes are "one-off" classes that have no name (odd ducks :))
- they are defined and created at the same time - "inline" in some other class using new() construct
- cannot be re-used anywhere else

The main use:

- create one-off/dynamic classes that are only needed in a specific context and do not "deserve" to have a separate class defined
- anonymous inner classes can either implement an interface or extend a class, but they can't do both at the same time
- they are not independent classes - they are either:
  - sub-classes of a concrete/abstract class OR
  - anonymous implementations of an interface



Ref: http://cs-fundamentals.com/java-programming/java-anonymous-inner-classes.php
https://www.geeksforgeeks.org/anonymous-inner-class-java/

@Marina Popova, Edward Sumitra

Anonymous classes - from Interfaces

Let's create a new class, AnimalManager, for more advanced functionality and testing

```
public class AnimalManager {

    public AnimalManager() {}

    public static ActionResult trainForTricks(ITrainable animalToTrain, String trickName) {
        return animalToTrain.doTrick(trickName);
    }

    public static ActionResult setupPlaydate(AbstractAnimal hostAnimal, AbstractAnimal friend) {
        return hostAnimal.playWithMe(friend);
    }
```

Anonymous classes - from Interfaces

And let's create one regular unit test, using concrete classes, and one "special" one, using an anonymous class implementing ITrainable interface with a modified behavior of the *doTrick()* method

```
@Test
public void testDoTrick() {
    ActionResult result = AnimalManager.trainForTricks(sneaky, trickName);
    assertEquals(ActionResult.FAILURE, result);

    result = AnimalManager.trainForTricks(bolt, trickName);
    assertEquals(ActionResult.SUCCESS, result);
}
```

body of the new class!

```
@Test
public void testDoTrick_anonimous_from_interface() {
    ActionResult result = AnimalManager.trainForTricks(
            new ITrainable() {
                public ActionResult doTrick(String trickName) {
                    System.out.println("I always do tricks!");
                    return ActionResult.SUCCESS;
                }
            },
            trickName);
    assertEquals(ActionResult.SUCCESS, result);
}
```

Anonymous classes - from classes

Let's create a one-off Cat-like class with modified implementation of the playWithMe() method. This anonymous class will extend the Cat class

```
@Test
public void testSetupPlaydate_anonimous_class() {
    // lets create a Special cat who will play with the dog
    // regardless of being friendly or not
    ActionResult result = AnimalManager.setupPlaydate(
            new Cat() {
                public ActionResult playWithMe(AbstractAnimal aFriend) {
                    if (isFriendly) {
                        System.out.println(name + " says: I'm friendly, I am playing with " +
                                aFriend.getName());
                    } else {
                        System.out.println(name + " says: I'm NOT friendly, but still playing with " +
                                aFriend.getName());
                    }
                    return ActionResult.SUCCESS;
                }
            },
            bolt);
    assertEquals(ActionResult.SUCCESS, result);
}
```

Java Unit testing

We are using the latest JUnit framework - JUnit 5

will discuss it in the Lab

https://junit.org/junit5/docs/current/user-guide/