Java SE 7

Module 2
Java OOP

LUXOFT TRAINING

LXFT LISTED NYSE

Java File Structure, Compilation and Running

Structure of the Java class

```
<access modifier> class <NameCamelCase>
{
   <fields>
   <methods>
}
```

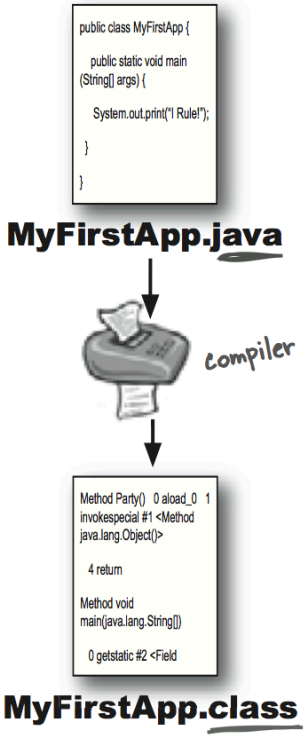
- **<access modifier>** defines class visibility, i.e. classes of which packages can access this class.
- The **public** modifier defines that such a class can be accessed from anywhere.
- Only one **public** class can be defined in a file.

Simplest Java App


```
public class MyFirstApp
{
    public static void main(String[] args)
    {
        System.out.println("I Rule!");
        System.out.println("The World");
    }
}
```

* Method **main** is an entry point for any Java application.

Running Java Application




```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("I Rule!");
     System.out.println("The World");
   }

}
```

1 Save

`MyFirstApp.java`

2 Compile

```
javac MyFirstApp.java
```

3 Run

```
File Edit Window Help Scream
%java MyFirstApp
I Rule!
The World
```

Command Line Arguments

What this **args** for?

```
public static void main(String[] args)
{
    …
}
```

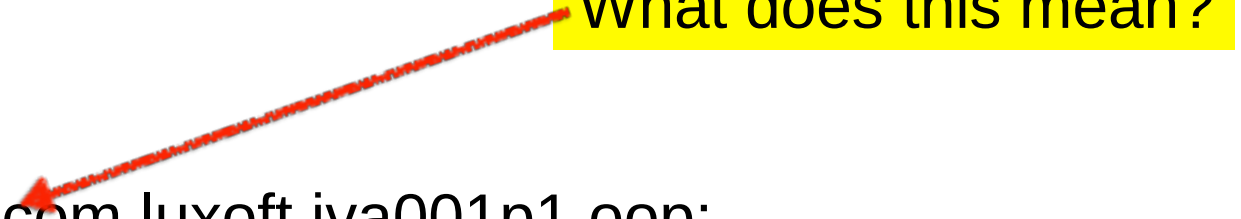
Command Line Arguments


```
public static void main(String[] args)
{
    String name = "I";

    if (args.length > 0)
    {
        name = args[0];
    }

    System.out.println(name + " Rule!");
    System.out.println("The World");
}
```

Quick Task

Use system command line to **compile** and **run** the program with different arguments.

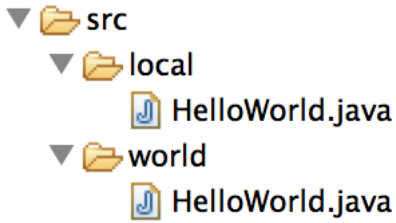
Package




```
package com.luxoft.jva001p1.oop;

public class E0_MyFirstApp
{
    public static void main(String[] args)
    {
        …
    }
}
```

Package

- Package is a directory.
- Used to avoid name collisions.
- Used to organize classes.

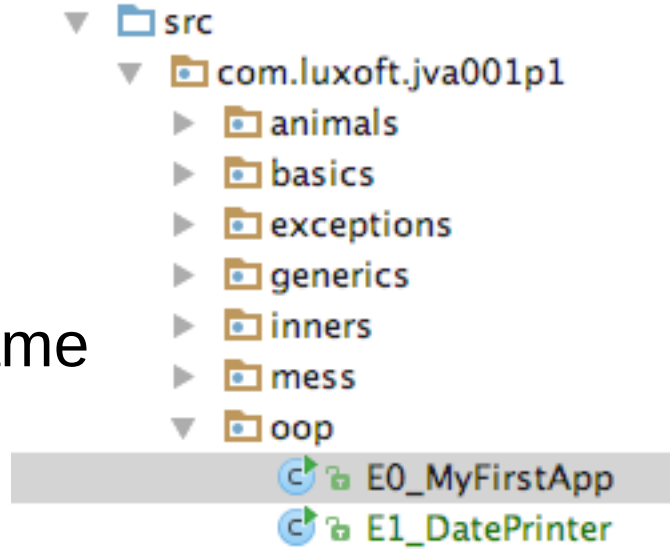


Just like two different **readme.txt** files can be located in different directories, java classes can be located in different directories as well.

Package

package com.luxoft.jva001p1.oop;

- It is necessary to specify package name in the file with keyword **package**.
- Package declaration should be the first statement.
- Dot (.) separator should be used to describe package path.



Import

To use a class located in some package you should specify its **full** name (pakage name + class name):

```
java.util.Date currentDate = new java.util.Date();
```


```
import java.util.Date;

public class E1_DatePrinter
{
    public static void main(String[] args)
    {
        Date currentDate = new Date();
    }
}
```

Import

- Class imports have nothing to do with loading imported class and doesn’t affect the compilation and running time.

- We can also include all classes of the given package to namespace: **import** java.util.*;

The Classpath concept

Directory(ies) where your ***.class** files located.

Used when you compile or run Java application with **javac** and **java** commands.

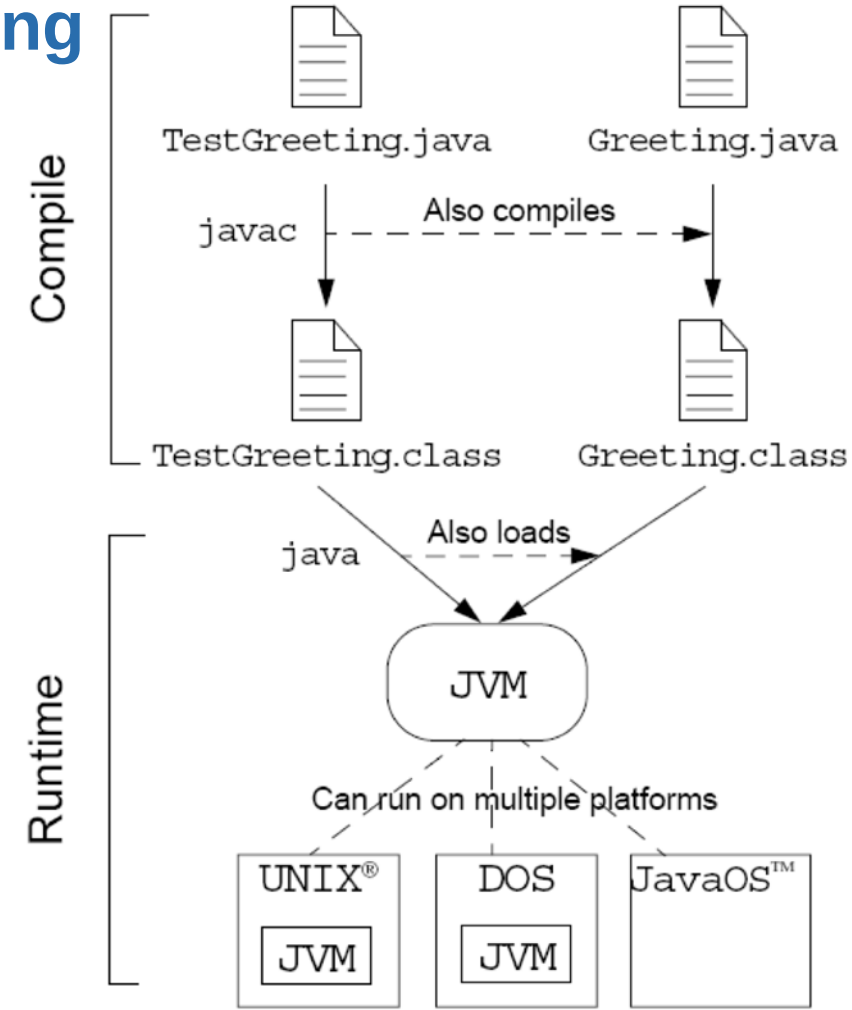
The Classpath concept

- Directories in Windows are separated by **";"**, in Unix by **“:"**

- If your **Person.java** file uses some **compiled** classes located in **c:\lib\classes** it is necessary to indicate the path for compiler through **-classpath** flag.

```
javac -classpath c:\lib\classes Person.java
```

Compiling and Running



Jar archives

- There is a way to organize classes in **zip archive** for more convenient propagation of the **.class** file group.

- The archive has **.jar** (java archive) extension.

- If classes are located in a ***.jar** file, it is necessary to specify the archive file name in **classpath**.

java –classpath c:\lib**myjar.jar** the.package.ClassToRun

Create *.jar

```
jar cf myjar.jar MainClass.class
```

c - creates *.jar file.

f - specifies file name.

Run Java application with runnable *.jar

Include into this jar special manifest ***.mf** file that describes where the class with **static main method** located.

- By default, this file created by the **jar tool**.
- The manifest is stored in the **META-INF** archive directory.
- You can also manually define a special **MANIFEST.MF** file in a JAR file.

Runnable *.jar

Manifest indicates the class name that contains main method.

Manifest-Version: 1.0

Main-Class: my.package.MyClass

Run this using: **java** -jar MyJar.jar

Quick Task

Create runnable jar to launch your program from Task 0.

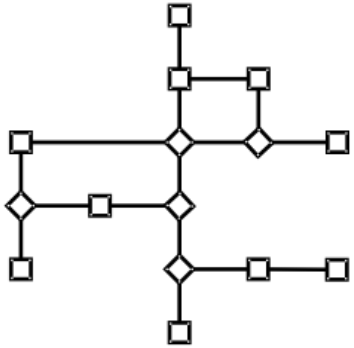
Programming paradigms

Programming paradigms

A programming paradigm is a collection of ideas and concepts defining a style of computer programming.

- Structured
- Logic
- Functional
- Object-oriented
- Aspect-oriented
- etc.

Structured Programming



- Describes the computation process as instructions that change program state.

- Very similar to imperative orders of a natural language, i.e. it is a sequence of instructions that a computer has to execute.

- Example: C language.

Logic Programming

- Based on formal logic.

- Usually specify what has to be computed, rather than the way the computation must be done.

- Example: Prolog.

Finding the factorial:

```
fact(1,1).
fact(N,F) :- N>1, N1 is N-1,
fact(N1,F1), F is F1*N.
?- fact(5,X).
X=120
```

Functional Programming

- Treats computation results as the evaluation of mathematical functions.

- Any functions is a superposition of other functions.

- Example: Lisp, Haskel, Closure.

- Support for functional programming: Java 8

Object Oriented Programming (OOP)

- Basic concepts are objects and classes.

- Appeared as the result of procedural programming evolution, when it was suggested to combine data and methods into OOP classes.

- Method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class.

Object Oriented Programming (OOP)

- Most common programming paradigm.

- Java is a fully OOP language, in other words, it doesn’t support procedural programming style.

Object Oriented Programming (OOP)

When you get objects from real world and describe it with programming language.

```
public class Cat
{
    private String name;

    public void voice()
    {
        System.out.println("meow");
    }

    public String getAddress() { return “Kiev"; }

    public void setName(String name) { this.name = name; }

    public String getName() { return name; }
}
```

Object concept

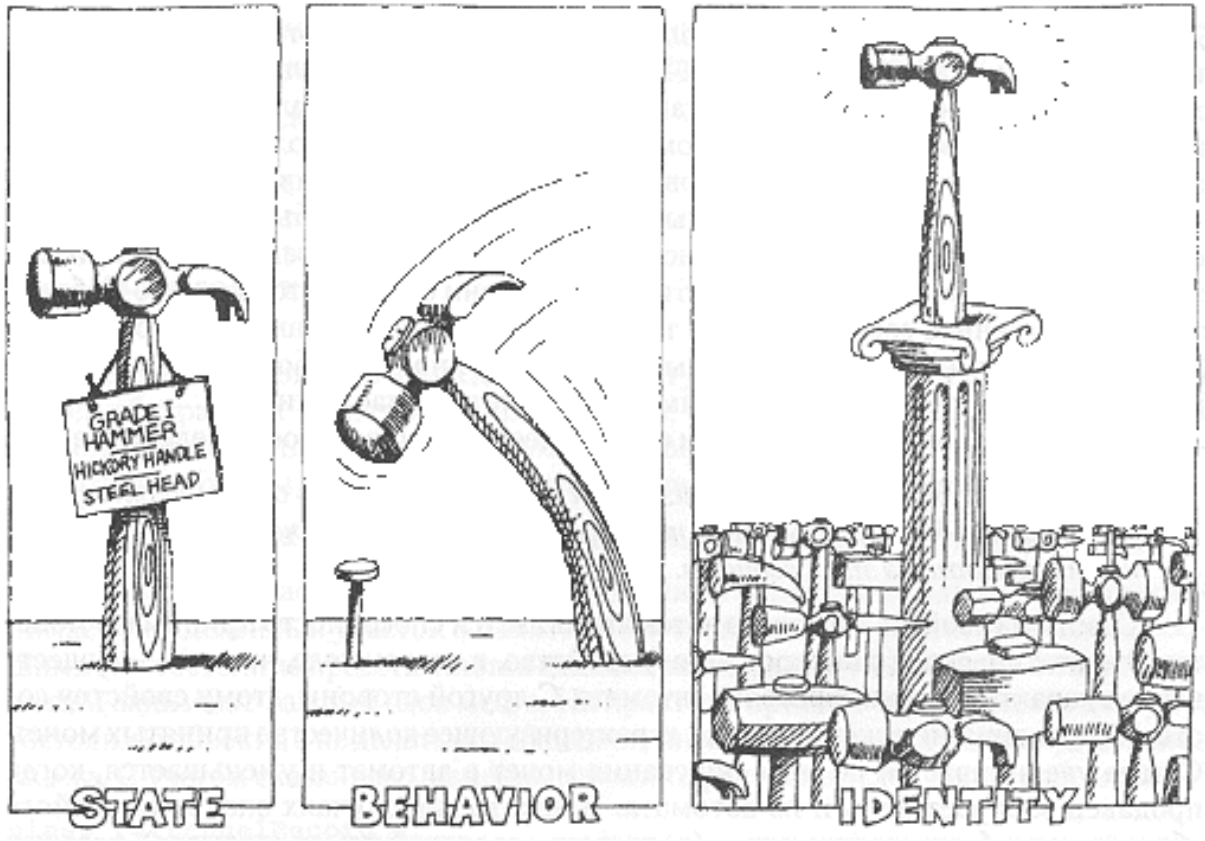
Classes and objects

- A **class** is an OOP language structure that represents a prototype used for creating instances of the class.

- An **object** is a tangible entity that exhibits some well-defined behavior and plays an important role in the domain.

- An object may receive messages.

What is object and what is not?



Classes and objects

- A **class** encapsulates state and behavior of real world object (concept) it is modeling.

- A **class** encapsulates the **state** with the help of data which is called **attributes** (class **properties**).

- As far as a class is an object prototype, when creating an object of a given class (corresponding to the prototype) a copy of data defined by classes is created in the memory.

Classes and objects

- A **class** encapsulates behavior with the help of code snippets that handle the state.

- This code snippets are called class **methods**.

- Sometimes it is convenient when class data is not duplicated at creation of each instance, but pertains to the class itself.

Classes and objects

When you get an object from real world and describe it with programming language.

```
public class Cat
{
    private String name;

    public void voice()
    {
        System.out.println("meow");
    }

    public String getAddress() { return “Kiev"; }

    public void setName(String name) { this.name = name; }

    public String getName() { return name; }
}
```

Quick Task

Look around and find some objects.

Quick Task

Objects demo.

Local and instance variables

1 **Instance** variables are declared inside a class but not within a method.

```
class Horse {
   private double height = 15.2;
   private String breed;
   // more code...
}
```

2 **Local** variables are declared within a method.

```
class AddThing {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

3 **Local** variables MUST be initialized before use!

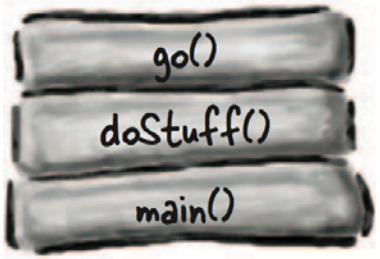
```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.

```
File Edit Window Help Yikes
% javac Foo.java
Foo.java:4: variable x might not have been initialized
      int z = x + 3;
1 error       ^
```

The Stack

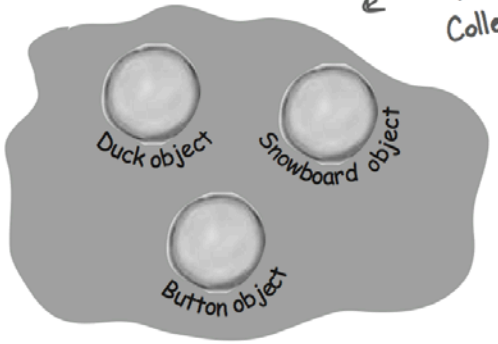
Where method invocations and local variables live



The Heap

Where ***ALL*** objects live

also known as "The Garbage-Collectible Heap"



Instance Variables

Instance variables are declared inside a *class* but *not* inside a method. They represent the "fields" that each individual object has (which can be filled with different values for each instance of the class). Instance variables live inside the object they belong to.

```
public class Duck {

    int size;

}
```

Every Duck has a "size" instance variable.

Local Variables

Local variables are declared inside a *method,* including method parameters. They're temporary, and live only as long as the method is on the stack (in other words, as long as the method has not reached the closing curly brace).

```
public void foo(int x) {

    int i = x + 3;

    boolean b = true;

}
```

The parameter x and the variables i and b are all local variables.

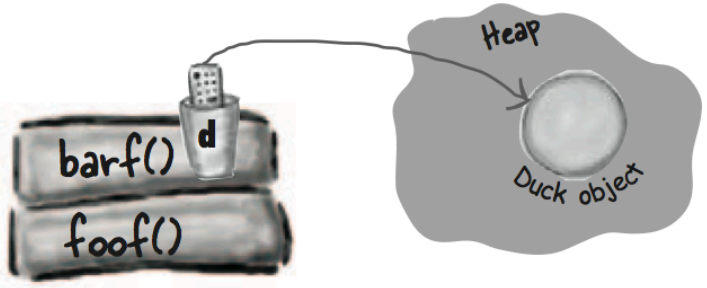
Object killer #1

Reference goes
out of scope, permanently.

```
public class StackRef
{
    public void foof()
    {
        barf();
    }

    public void barf()
    {
        Duck duck = new Duck();
    }
}
```

1. *foof()* is pushed onto the Stack, no variables are declared.

2. *barf()* is pushed onto the Stack, where it declares a reference variable, and creates a new object assigned to that reference. The object is created on the Heap, and the reference is alive and in scope.



3. *barf()* completes and pops off the Stack. Its frame disintegrates, so 'd' is now dead and gone. Execution returns to *foof()*, but *foof()* can't use 'd'.

Object killer #2

Assign the reference to another object.

```
public class ReRef
{
    private Duck duck = new Duck();

    public void go()
    {
        duck = new Duck();
    }
}
```

Object killer #3

Explicitly set the reference to null.

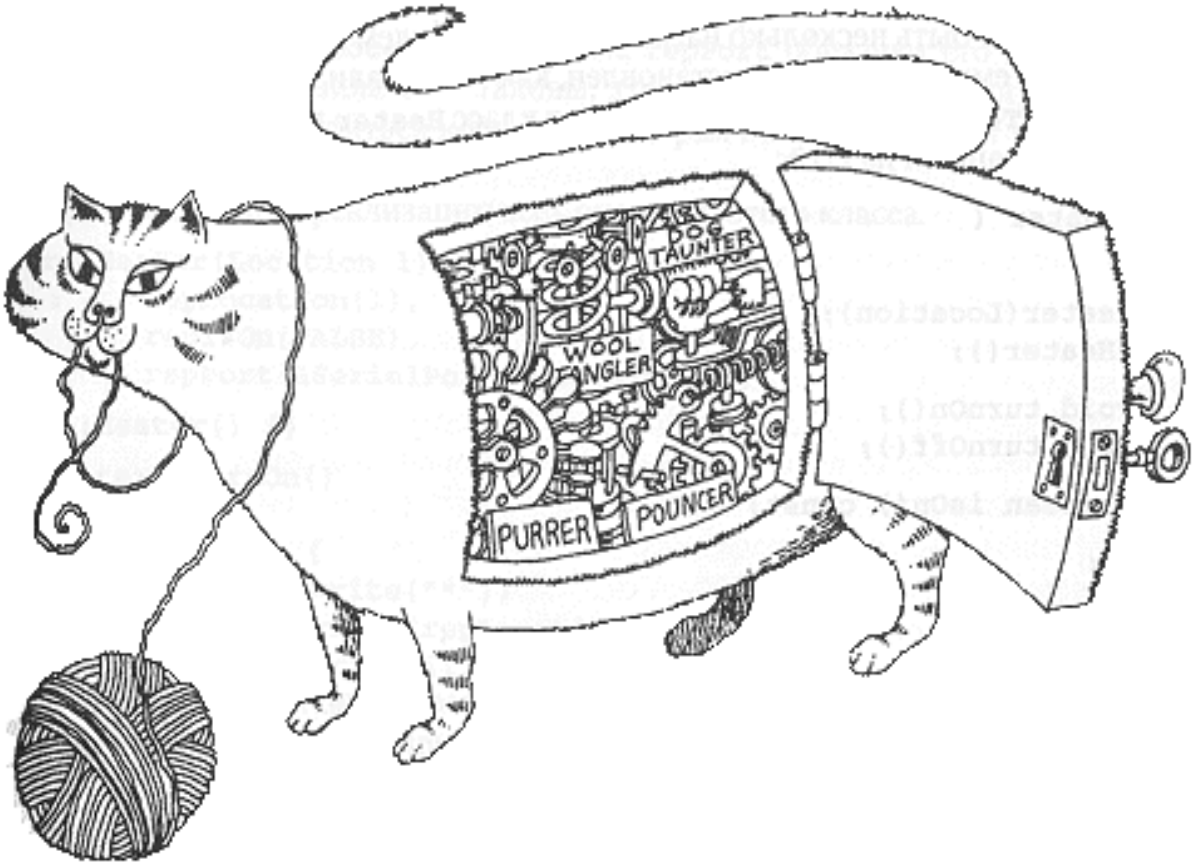
```
public class ReRef
{
    private Duck duck = new Duck();

    public void go()
    {
        duck = null;
    }
}
```

Tasks

- Exercise 1

The three most important words in OOP

Encapsulation



“Black Box” concept

Encapsulation

- Encapsulation is a language mechanism that restricts the access to object components.

- Hiding implementation details behind allowable messages.

- Encapsulated variables can be accessed when writing class implementation.

Encapsulation

Encapsulation allows **not to depend** on the inner structure of the object.

Examples:

- Driving a car
- Doing a job by specialist

It allows to change the inside structure and mechanisms of the object using an interface.

Encapsulation

- Encapsulation - **do not** related to security.

- Encapsulation - related to **conrol**.

It is a mechanism to make the code more easy to understand and reuse, concentrate on big picture instead of details.

Encapsulation

How **encapsulation** looks like?

Encapsulation

```
public class Cat
{
    private String name;

    protected String address;

    public void voice() { System.out.println(“meow"); }

    public String getName() { return name; }
}
```

Encapsulation

We can control access to: **fields** and **methods**.

private – hide from everyone, only this class has the access.

default – hide from everyone except for this class and classes from same package.

protected – hide from everyone except for this class, classes from same package, and this class descendants.

public – everyone can get an access.

Java Beans

```
public class Clock
{
    private String time;

    public Clock() { }

    public String getTime() { return time; }

    public void setTime(String time) { this.time = time; }

    public static void main(String[] args)
    {
        Clock clock = new Clock();

        clock.setTime("12:30");

        String time = clock.getTime();
        System.out.println(time);
    }
}
```

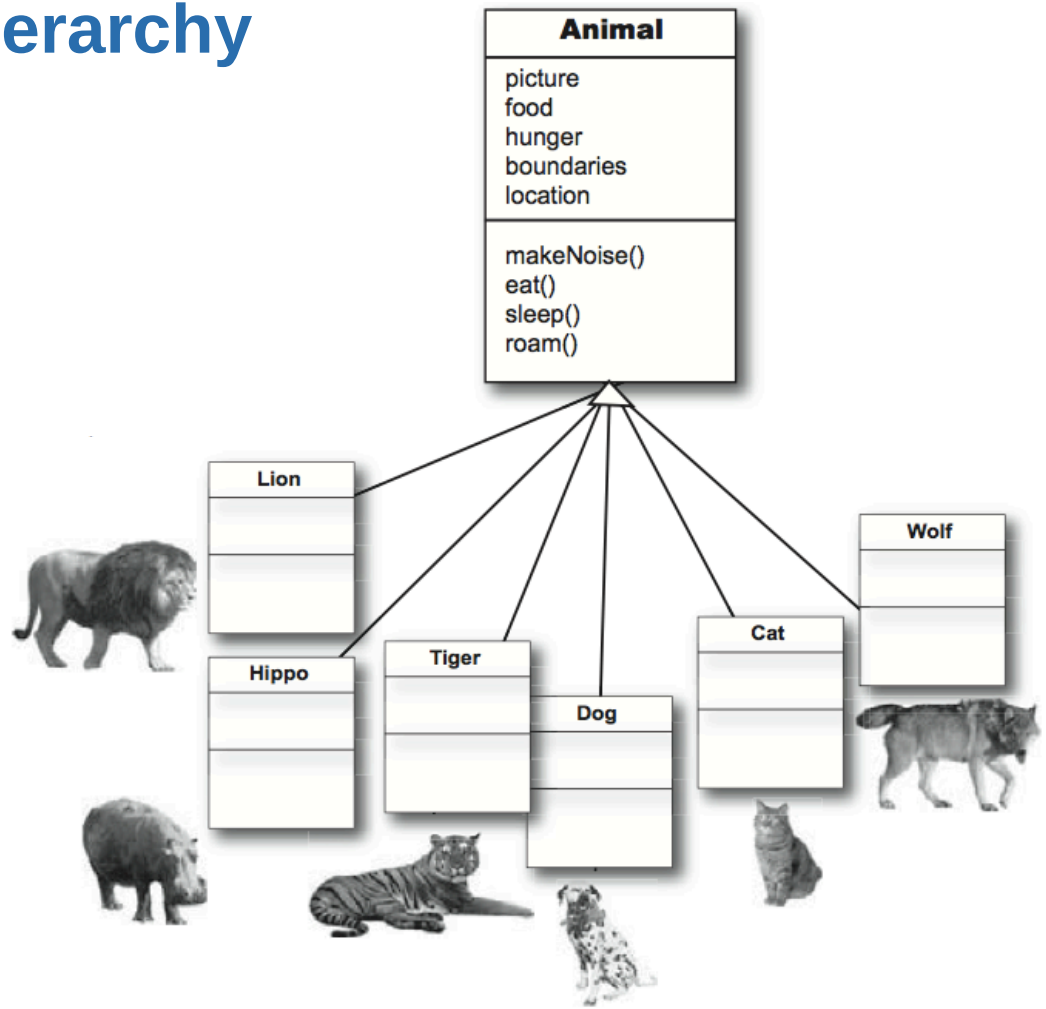
Inheritance

1. Look for objects that have common attributes and behaviors.

What do these six types have in common? This helps you to abstract out behaviors. (step 2)

How are these types related? This helps you to define the inheritance tree relationships (step 4-5)

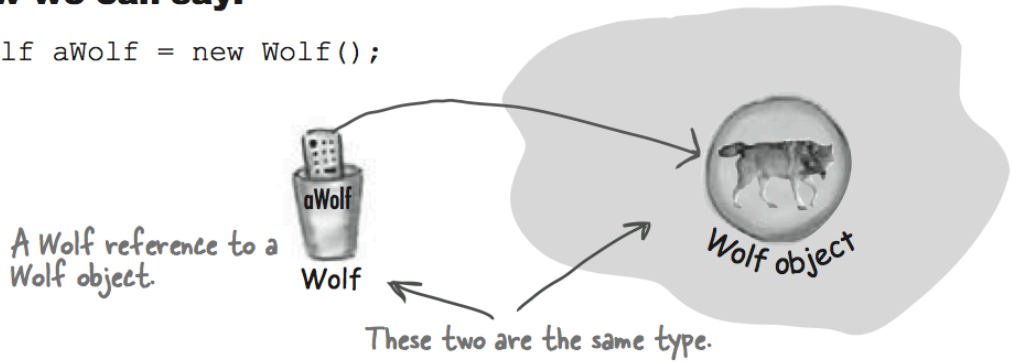
Inheritance hierarchy



Inheritance hierarchy

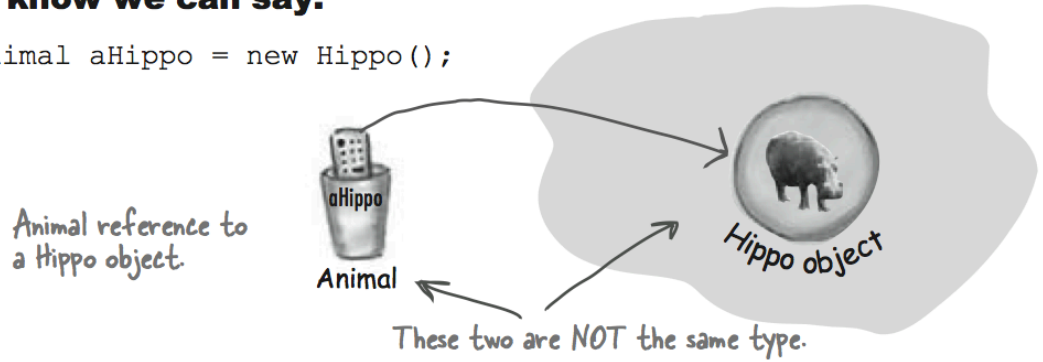
We know we can say:

```
Wolf aWolf = new Wolf();
```



And we know we can say:

```
Animal aHippo = new Hippo();
```



Constructors

- It should be ensured that the child constructor will be called after the parent constructor.

- The **super**(arg1, …) keyword is used to call superclass constructor.

- The needed constructor is selected by the list of the arguments.

Constructors

The **super**(arg1, …) keyword is used to call the superclass constructor.

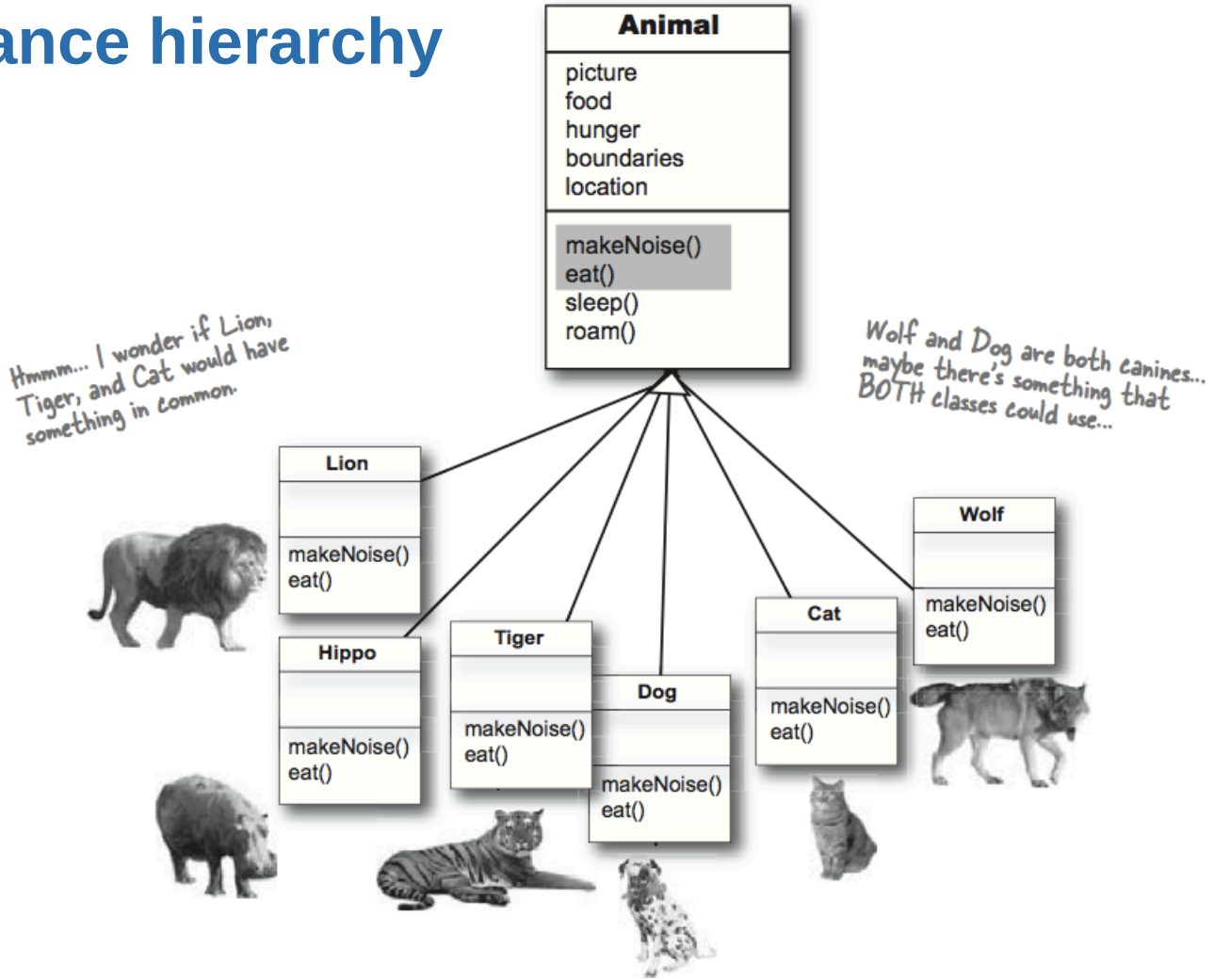
```
public class SuperClass
{
    public SuperClass(int foo) { }
}

public class SubClass extends SuperClass
{
    public SubClass(int foo, int bar)
    {
        super(foo);
    }
}
```

Constructors

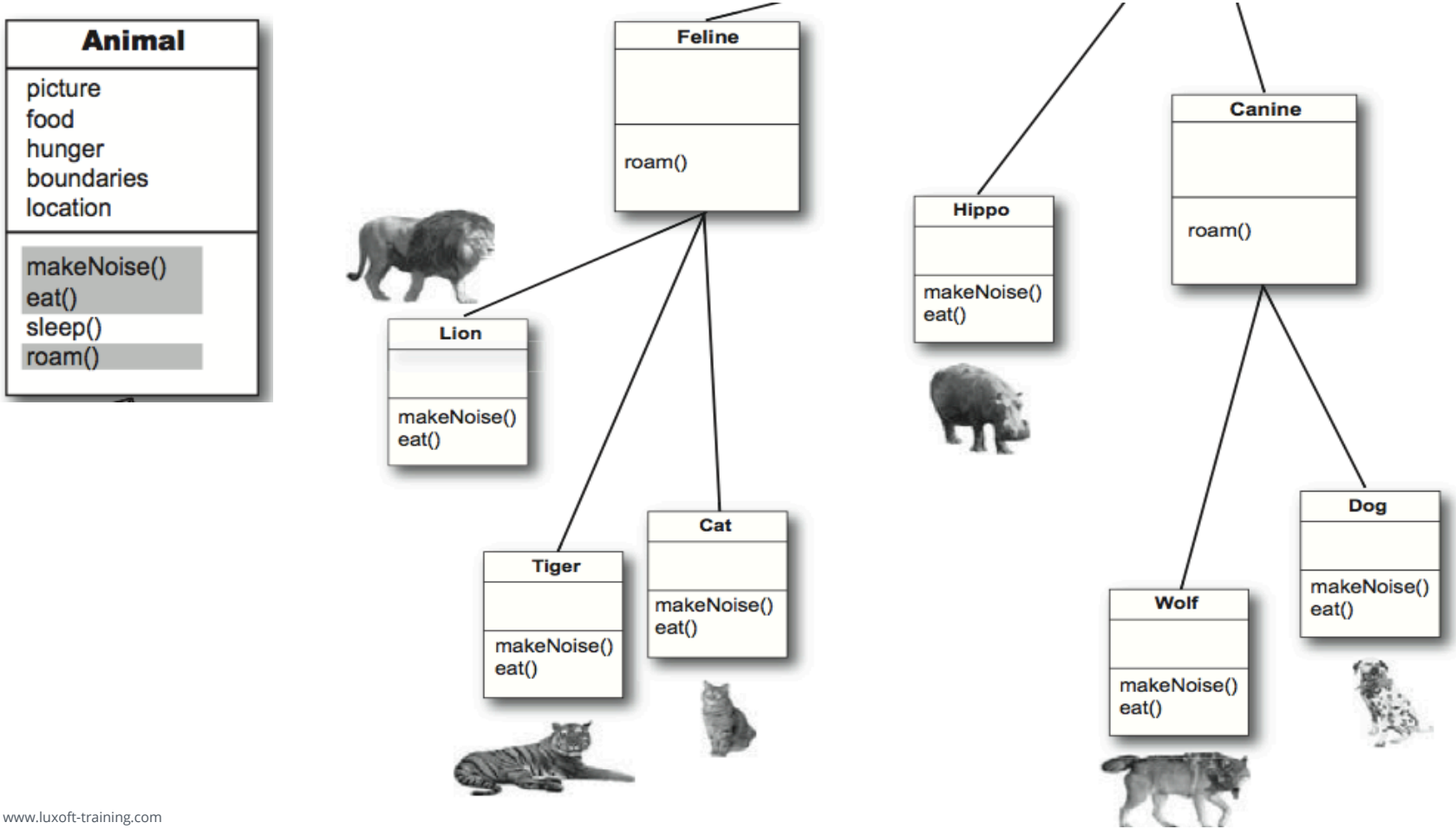
The **this**(arg1, …) reference can be used to call overloaded constructor.

```
public Employee(String name)
{
    this(name, 23);
}

public Employee(String name, int age) { }
```

Inheritance hierarchy

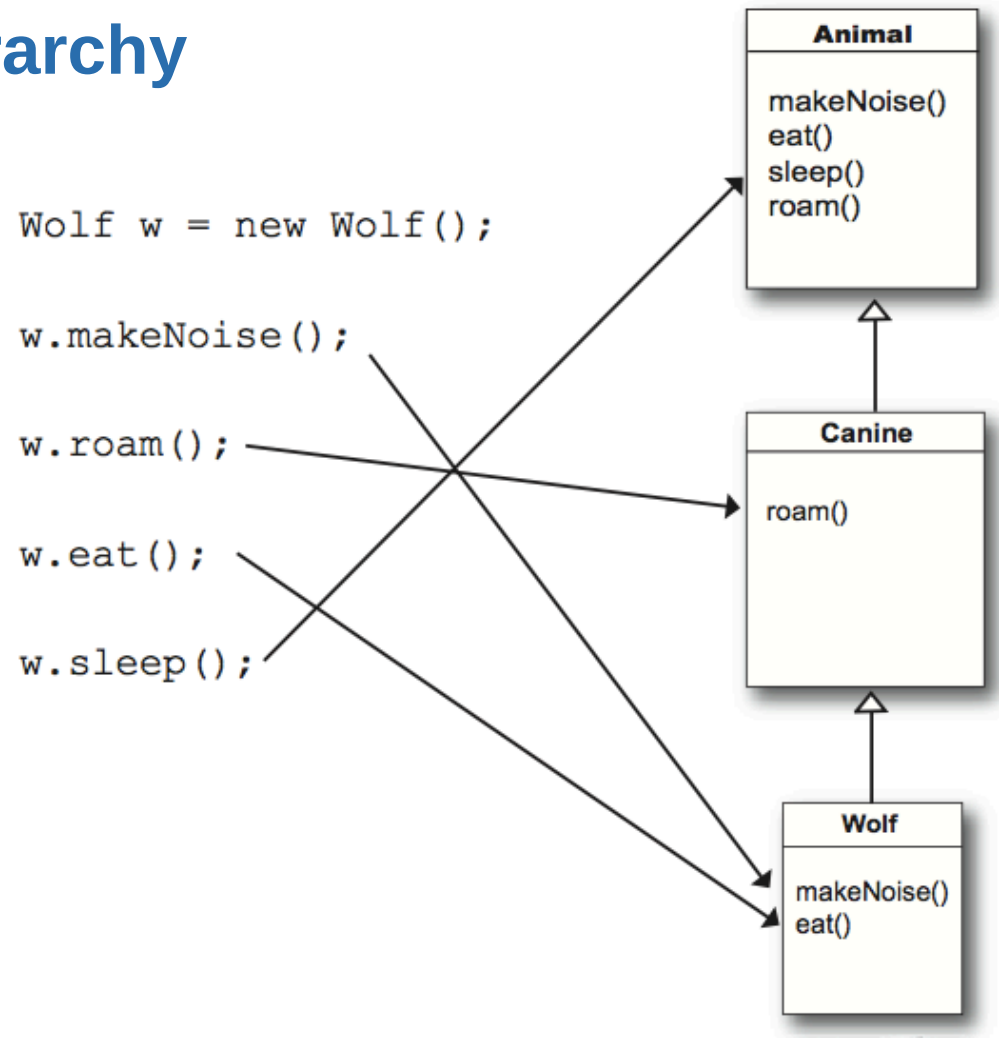


Inheritance hierarchy



Inheritance hierarchy


```
Wolf w = new Wolf();

w.makeNoise();

w.roam();

w.eat();

w.sleep();
```



Method Overridng

Overriding method must:

- Have the same name and arguments list as parent class method.

- The return type should be the same or its subclass.

Method Overridng

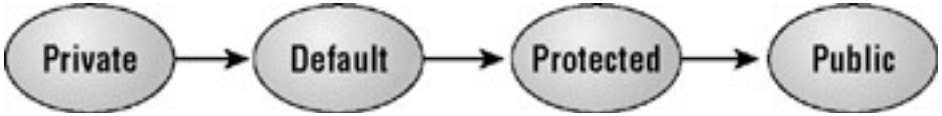
Requirements to overridden method:

- **final** cannot be overridden.

- Access modifier shall not be narrower.

- Overridden method should throw checked exceptions of the same type or subclass.

Inheritance & Access Modifiers

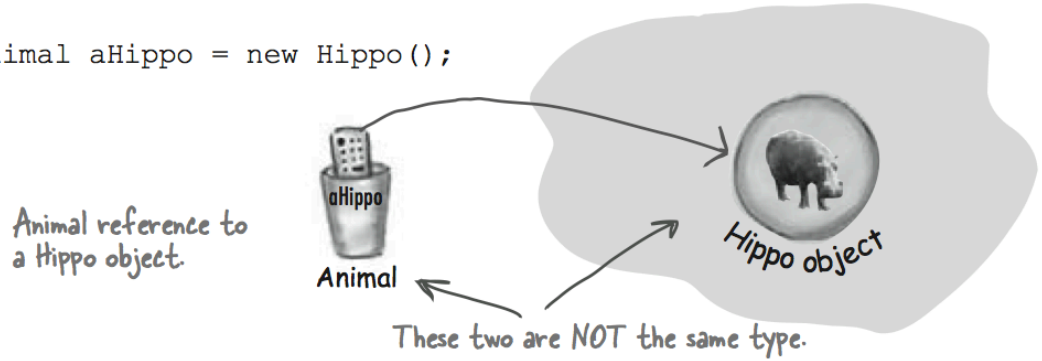


- When overriding a method you **can't** make its scope **less visible**.

- Visibility scope of overridden method **can be the same** as visibility scope of an overriding method or **even higher**.

We can do like this because of Polymorphism

```
Animal aHippo = new Hippo();
```



Every class is Object

① equals(Object o)

```
Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println("true");
} else {
    System.out.println("false");
}
```

```
File Edit Window Help Stop
% java TestObject
false
```

Tells you if two objects are considered 'equal' (we'll talk about what 'equal' really means in appendix B).

② getClass()

```
Cat c = new Cat();
System.out.println(c.getClass());
```

```
File Edit Window Help Faint
% java TestObject
class Cat
```

Gives you back the class that object was instantiated from.

③ hashCode()

```
Cat c = new Cat();
System.out.println(c.hashCode());
```

```
File Edit Window Help Drop
% java TestObject
8202111
```

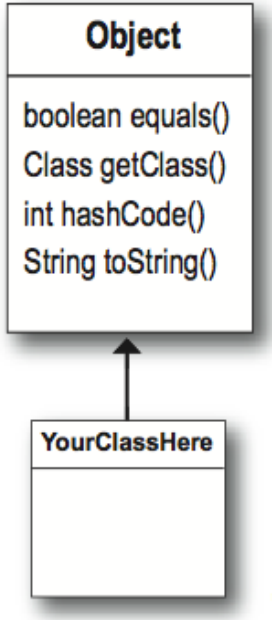
Prints out a hashcode for the object (for now, think of it as a unique ID).

④ toString()

```
Cat c = new Cat();
System.out.println(c.toString());
```

```
File Edit Window Help LapseIntoComa
% java TestObject
Cat@7d277f
```

Prints out a String message with the name of the class and some other number we rarely care about.



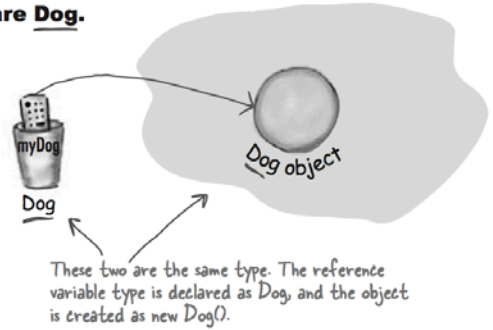
Polymorphism

Polymorphism is a possibility of the objects of the same type to have different behavior.

Using Polymorphism

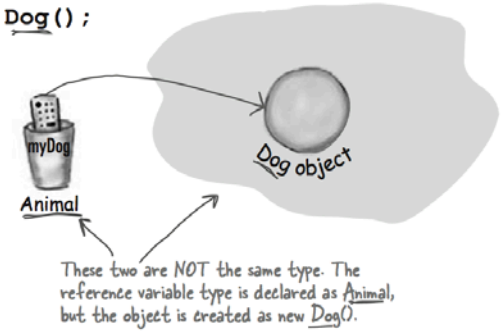
The important point is that the reference type AND the object type are the same.

In this example, both are Dog.



But with polymorphism, the reference and the object can be *different*.

```
Animal myDog = new Dog();
```

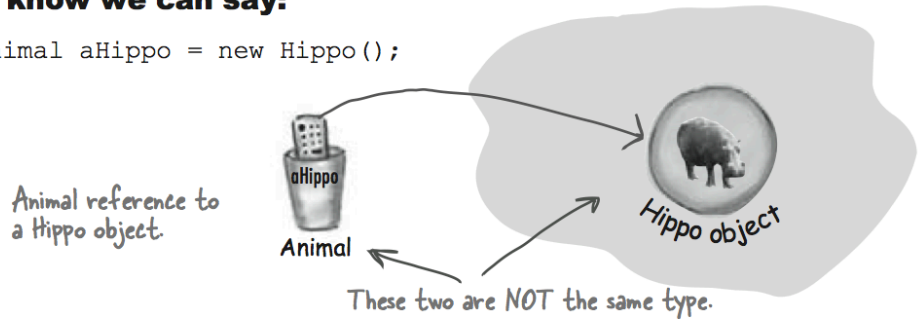



```
Animal[] animals = new Animal[5];

animals [0] = new Dog();

animals [1] = new Cat();

animals [2] = new Wolf();

animals [3] = new Hippo()

animals [4] = new Lion();


for (int i = 0; i < animals.length; i++) {


    animals[i].eat();


    animals[i].roam();

}
```

Abstract Class

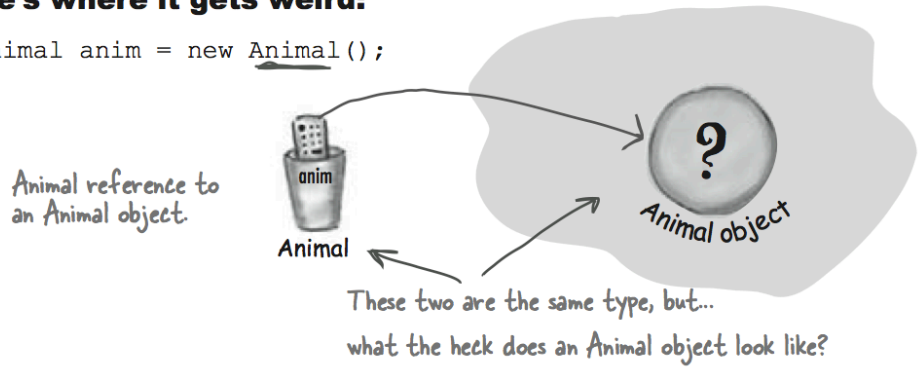
And we know we can say:

```
Animal aHippo = new Hippo();
```

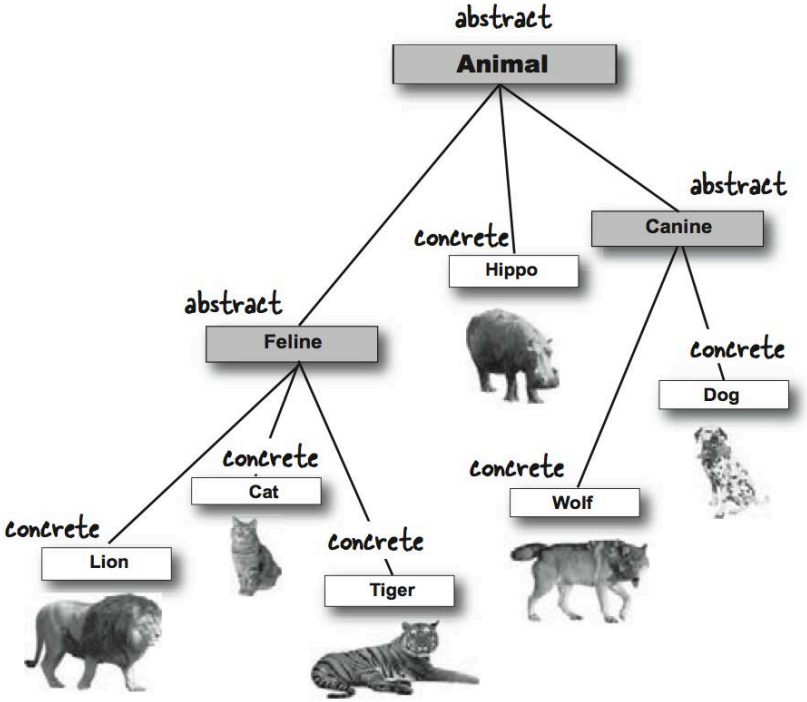


But here's where it gets weird:

```
Animal anim = new Animal();
```



Abstract Class

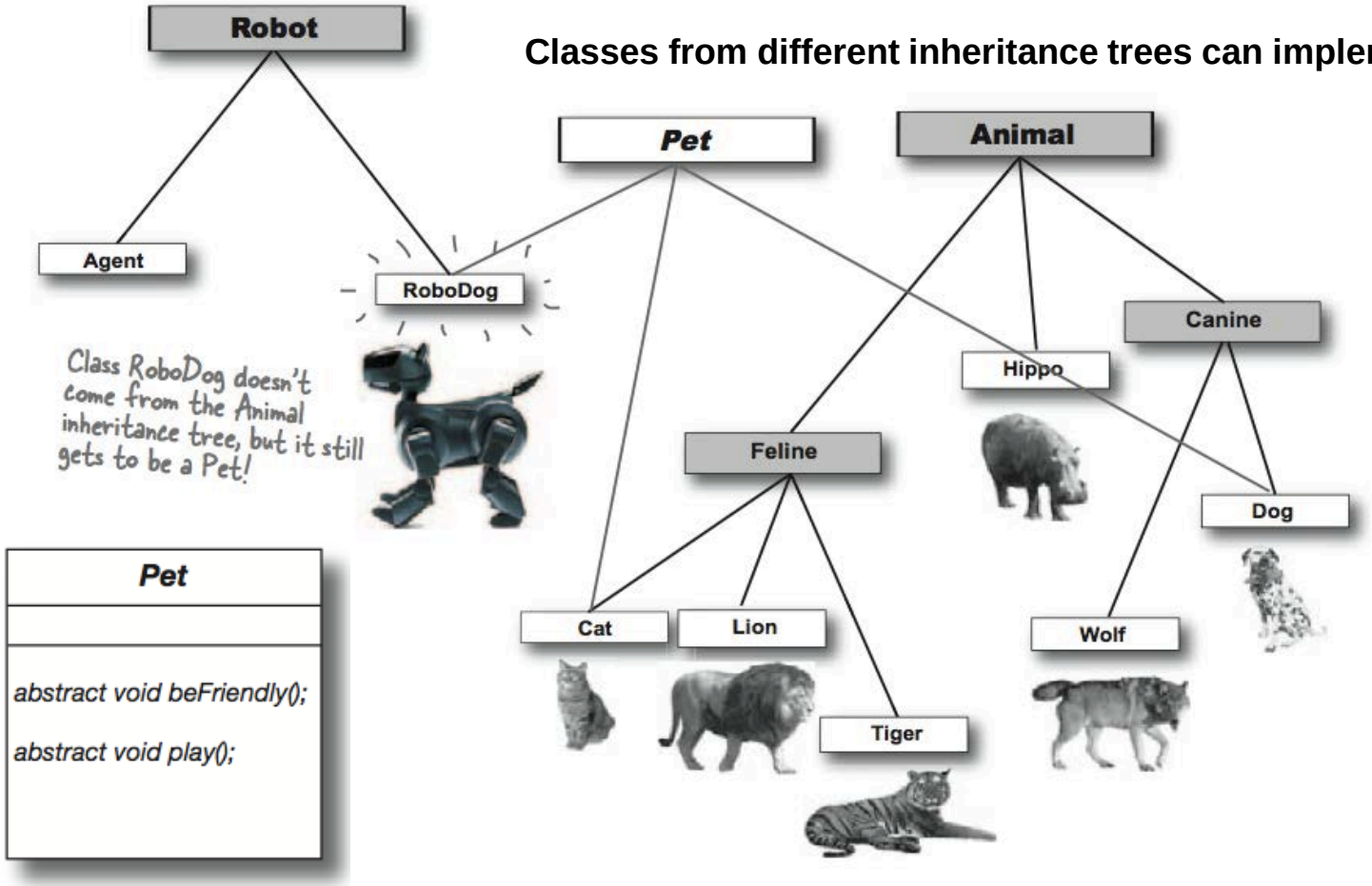



```
abstract class Canine extends Animal {
   public void roam() { }
}

public abstract void eat();
```

No method body!
End it with a semicolon.

Using Interfaces

Classes from different inheritance trees can implement the same interface.



Using Interfaces


```
public interface Pet
{
    void beFriendly();

    void play();
}

public class Dog extends Animal implements Pet
{
    @Override
    public void beFriendly()
    {
        System.out.println("brrr...");
    }

    @Override
    public void play()
    {
        System.out.println("Running with the ball...");
    }
}
```

Using Interfaces

Class can implement multiple interfaces:

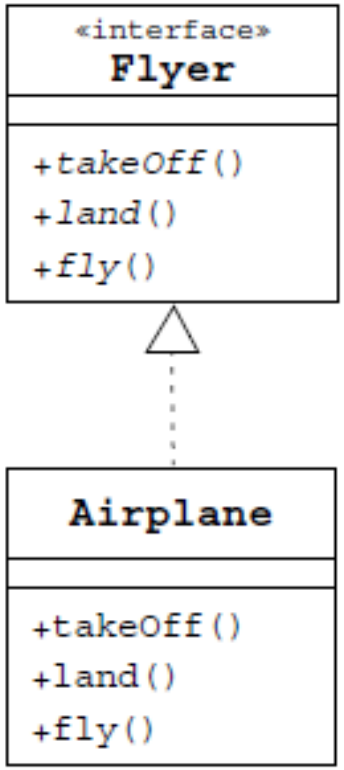
```
public class Dog extends Animal implements Pet, Saveable, Paintable {...}
```

Interfaces

- **Interface** forms a contract between the client code and a **class** that **implements** this **interface**.
- Java interface is declared with the **interface** keyword.
- A class that **implements** an **interface** contains the **implements** clause in the **class** declaration.
- An **interface** may be considered as an abstract class whose methods are all abstract.

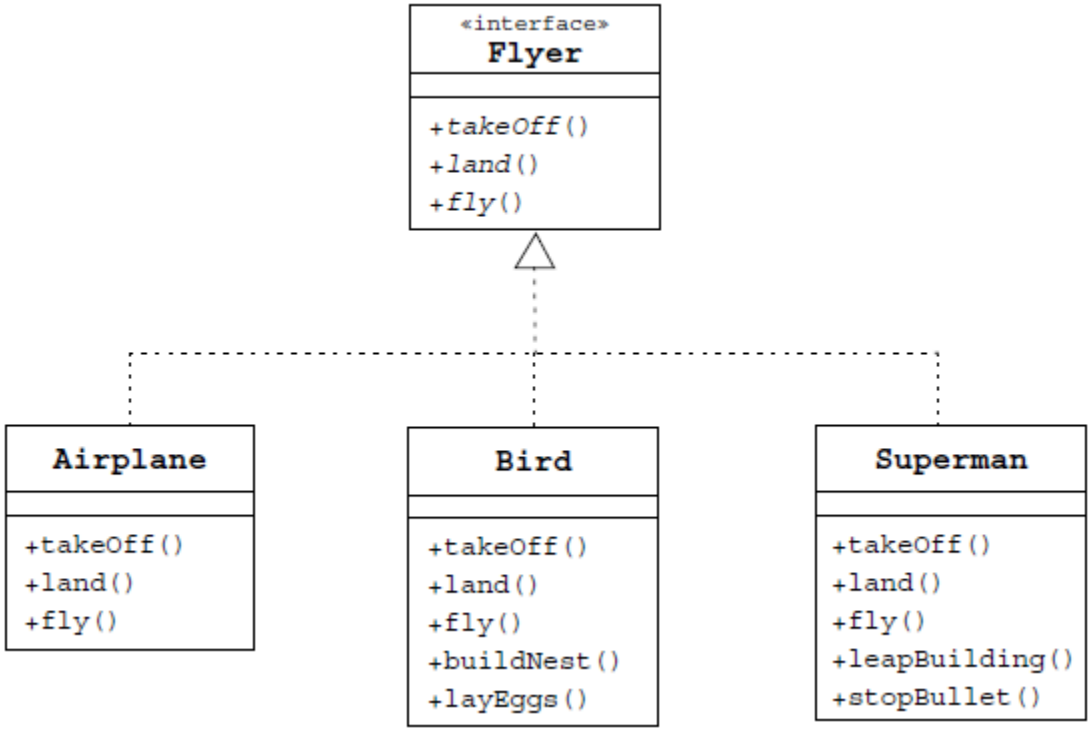
Interfaces


```
public interface Flyer
{
    void takeOff();

    void land();

    void fly();
}

public class Airplane implements Flyer
{
    @Override
    public void takeOff() {...}

    @Override
    public void land() {...}

    @Override
    public void fly() {...}
}
```



Interfaces

As long as an interface is a specification of certain behavior, the class can implement several interf



Interfaces


```
public interface Flyer
{
    void fly();
}
```

```
public interface Swimmer
{
    void swim();
}
```

```
public class Duck implements Flyer, Swimmer
{
    @Override
    public void fly() {...}

    @Override
    public void swim() {...}
}

public class Penguin implements Swimmer
{
    @Override
    public void swim()
    {
        // A penguin is able to swim, but not able to fly
    }
}
```

Interfaces

As long as an interface is a contract rather than an implementation:

- It cannot be instantiated
- There are no constructors
- There is no instance data

Interfaces

It is assumed that all interface methods are declared as **public abstract**

```
public interface Flyer
{
   void fly();
}

// This equals:
public interface Flyer
{
    public abstract void fly();
}
```

Interfaces

Note! An interface can contain **static final** data.

Interfaces

The **public static final** modifier is optional.

```
public interface Flyer
{
    public static final int WINGS = 2;

    void fly();
}

public interface Flyer
{
    // This equals:
    int WINGS = 2;

    void fly();
}
```

Interfaces

- An interface that is not nested cannot be **private**.
- An interface cannot be **protected**.
- A **public** interface can be implemented by any class.
- A default interface can be implemented by any class from the package that defines the interface itself.

Interfaces

When overriding the method you cannot reduce its visibility.

```
public interface Swimmer
{
    void swim();
}


public class Penguin implements Swimmer
{
    @Override
    protected void swim()
    {
        // Compiler error! Cannot reduce the visibility of the inherited method from Swimmer
    }
}
```

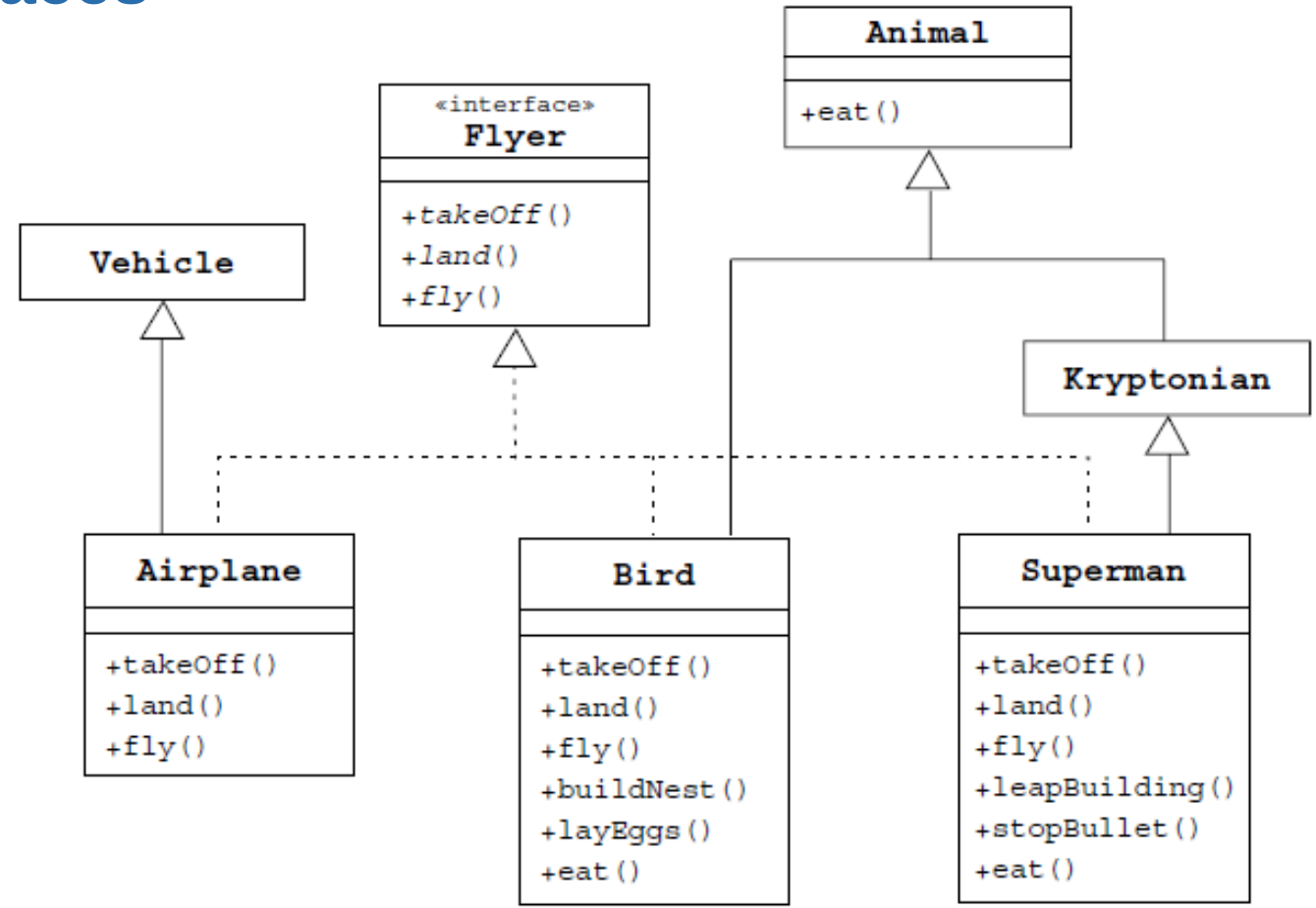
Interfaces

A behavior can extend another behavior.

```
public interface Set<E> extends Collection<E> {...}
```

Note! the class that implements an extended interface **must** implement methods of **both** interfaces.

Interfaces


Animal
+eat()
«interface»
Flyer
+takeOff()
+land()
+fly()
Vehicle
Kryptonian
Airplane
+takeOff()
+land()
+fly()
Bird
+takeOff()
+land()
+fly()
+buildNest()
+layEggs()
+eat()
Superman
+takeOff()
+land()
+fly()
+leapBuilding()
+stopBullet()
+eat()

Interfaces


```
public class Bird extends Animal implements Flyer
{

    @Override
    public void takeOff() {...}

    @Override
    void eat() { super.eat(); }

    @Override
    public void land() {...}

    @Override
    public void fly() {...}

    public void buildNest() {...}

    public void layEggs() {...}
}
```

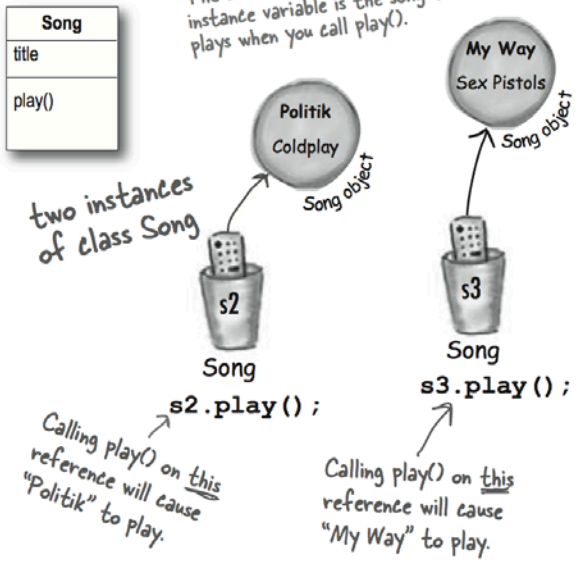
Static

regular (non-static) method

```
public class Song {
   String title;
   public Song(String t) {
     title = t;
   }
   public void play() {
      SoundPlayer player = new SoundPlayer();
      player.playSound(title);
   }
```

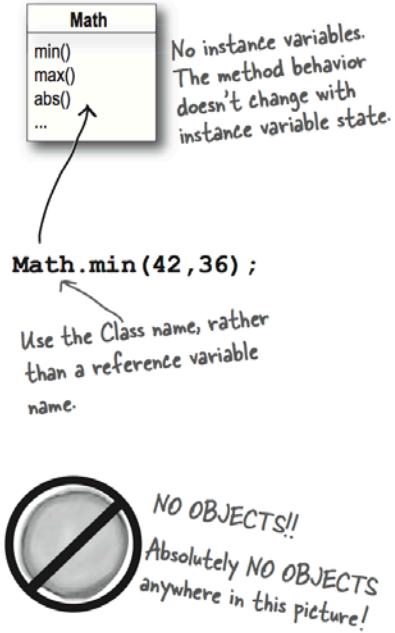
Instance variable value affects the behavior of the play() method.

The current value of the 'title' instance variable is the song that plays when you call play().

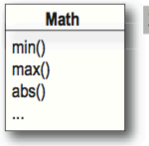


static method

```
public static int min(int a, int b){
    //returns the lesser of a and b
}
```



Call a static method using a class name




```
Math.min(88,86);
```

Call a non-static method using a reference variable name


```
Song t2 = new Song();
t2.play();
```

Static methods can't use non-static (instance) variables!

Static imports

Some old-fashioned code:

```
import java.lang.Math;

class NoStatic {

  public static void main(String [] args) {

    System.out.println("sqrt " + Math.sqrt(2.0));

    System.out.println("tan " + Math.tan(60));

  }

}
```

The syntax to use when declaring static imports.

Same code, with static imports:

```
import static java.lang.System.out;

import static java.lang.Math.*;

class WithStatic {

  public static void main(String [] args) {

    out.println("sqrt " + sqrt(2.0));

    out.println("tan " + tan(60));

  }

}
```

Static imports in action.

Constants

Initialize a *final* static variable:

① At the time you declare it:

```
public class Foo {
   public static final int FOO_X = 25;
}
```

notice the naming convention -- static final variables are constants, so the name should be all uppercase, with an underscore separating the words

OR

② In a static initializer:

```
public class Bar {
   public static final double BAR_SIGN;

   static {
        BAR_SIGN = (double) Math.random();
   }
}
```

this code runs as soon as the class is loaded, before any static method is called and even before any static variable can be used.

Final modifiers

final method

```
class Poof {
    final void calcWhuffie() {
       // important things
       // that must never be overridden
    }
}
```

final class

```
final class MyMostPerfectClass {
   // cannot be extended
}
```

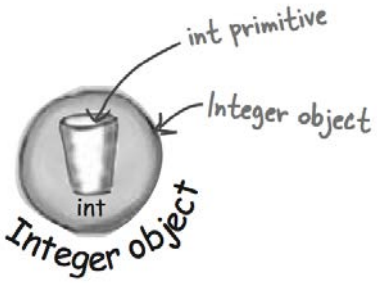
non-static final variables

```
class Foof {
    final int size = 3;
    final int whuffie;

    Foof() {
       whuffie = 42;
    }

    void doStuff(final int x) {
       // you can't change x
    }

    void doMore() {
       final int z = 7;
       // you can't change z
    }
}
```

now you can't change size

now you can't change whuffie

Wrappers



Boolean

Character

Byte

Short

Integer

Long

Float

Double

Watch out! The names aren't mapped exactly to the primitive types. The class names are fully spelled out.

Give the primitive to the wrapper constructor. That's it.

wrapping a value

```
int i = 288;
Integer iWrap = new Integer(i);
```

All the wrappers work like this. Boolean has a booleanValue(), Character has a charValue(), etc.

***un*wrapping a value**

```
int unWrapped = iWrap.intValue();
```

Autoboxing


```
private Integer i = 3;

public void doNumsNewWay()
{
    List<Integer> numbers = new ArrayList<>();
    numbers.add(3);

    int num = numbers.get(0);
}
```

Primitive Wrappers

Java provides wrapper classes for each primitive data types.

Wrappers are found in **package** java.lang

Byte, Short, Integer, Long, Float, Double, Character

You may use wrappers absolutely transparently:

```
Integer count   = 1;
Boolean isReady = false;
```

Primitive Wrappers

Java automatically converts an object to a primitive type if it is required.

The Boolean object will contain **true**, if the constructor parameter will be equal to **"True"** string in any case.

```
Boolean isReady = new Boolean("True"); // true

Boolean isYes = new Boolean("Yes"); // false
```

Primitive Wrappers

- Numeric types are inherited from the **Number** class.
- Each class has a set of constants with maximum and minimal values.
- Each class has static methods for converting type from string.

```
Integer.MIN_VALUE;
Integer.MAN_VALUE;
Double.parseDouble("34.1");
```

Primitive Wrappers

The **Double** class has the method of checking whether the number is infinite.

```
Double.isInfinite(23.9);
```

The **Integer** class has useful methods for work with binary representation of integers.

```
Integer.reverse(23);
Integer.bitCount(23);
Integer.numberOfLeadingZeros(23);
```

Primitive Wrappers

To work with bigger numbers, classes from the **java.math** package can be used:

BigInteger

BigDecimal

These classes store numbers as strings.

```
BigInteger number = new BigInteger("33");
BigInteger big = number.pow(10000);
```

Tasks

- Exercise 2 Bank Application

Tasks (optional)

- Exercise 3

Enumerations

Very often we have to introduce enumerated types:

```
public class PlayingCard
{
   public static final int SUIT_SPADES = 0;
   public static final int SUIT_HEARTS = 1;
   public static final int SUIT_CLUBS = 2;
   public static final int SUIT_DIAMONDS = 3;

   private int suit;

   public PlayingCard(int suit)
   {
      this.suit = suit;
   }

   public String getSuitName()
   {
      String name = "";

      switch (suit)
      {
         case SUIT_SPADES: name = "spades"; break;
         case SUIT_HEARTS: name = "hearts"; break;
         case SUIT_CLUBS: name = "clubs"; break;
         case SUIT_DIAMONDS: name = "diamonds"; break;
      }
      return name;
   }
}
```

Enumerations

Enumeration is subclass of the **java.lang.Enum** class. Enumeration solves this problem and can be applied to the **switch** operator.

Enumeration is a usual class with some limitations.

Enumerations

- Declared with the help of **enum**.
- Enumeration instance cannot be explicitly created.
- Enumeration cannot be extended.
- Enumeration can be the **switch** argument.
- Has embedded **.name()** method that prints enumeration values.

Enumerations


```
public enum LightState
{
    RED, YELLOW, GREEN;
}

public static void main(String[] args)
{
    switch (nextTrafficLight.getState())
    {
        case LightState.RED; stop(); break;
        case LightState.YELLOW; floorIt(); break;
        case LightState.GREEN; go(); break;
    }
}
```

Enumerations

oop.enums.Suit

```
public enum Suit
{
    DIAMOND(true), HEART(true), CLUB(false), SPADE(false);

    private boolean red;

    Suit(boolean red)
    {
        this.red = red;
    }

    public boolean isRed()
    {
        return red;
    }

    @Override
    public String toString()
    {
        return name() + (red ? ":red" : ":black");
    }
}
```

Tasks (optional)

- Exercise 4 (file T1_Enums)