Java 4 WEB

Lesson 3 - OOP

Lesson goals

- Class design
- Implement encapsulation
- Implement inheritance including visibility modifiers and composition
- Implement polymorphism

Object vs Class

Dog

Class

Object

Object

Abstract Classes

- May contain any number of methods including zero
- If class has at least one abstract method – class is abstract
- **abstract void** clean()
- Abstract methods may not appear in a class that is not abstract
- The first concrete subclass of an abstract class is required to implement all abstract methods that were not implemented by a superclass

Method vs Constructor

Method describes behavior

Method signature:

- name
- arguments (including order)

Specific method with the same to class name and without return statement

called Constructor

Constructor

```
class Animal {
    String name;

    Animal() {
        this("No-Name");
    }

    public Animal(String name) {
        this.name = name;
    }
}
```

```
class Main{
    public static void main(String[] args) {
    println(new Monkey());
    println(new Monkey(2));
    }
}
```

```
class Monkey extends Animal {
    int paws;

    public Monkey() {
        super(); // not necessary
    }

    Monkey(int paws) {
        super("Bandar-log");
        paws = paws;    // BAD
        this.paws = paws;
    }

    public void Monkey() {
        // this(3);          // BAD
    }

    public void Monkey(int i) {
        System.out.println("OMG");
    }
}
```

Overloading and Overriding


```
class Game {
    void play() {
        // move to the left
    }
}
```

```
class Mario extends Game {
    @Override
    public void play() { // overriding
        // move to the left, move to the right
    }

    void play(Character character) { // overloading
        // omg, I'm chicken
    }

    int play() { // will not compile
        // lol
    }
}
```

Overriding rules

- The access modifier must be the same or more accessible;
- The return type must be the same or a more restrictive type, also known as covariant return types;
- If any checked exceptions are thrown, only the same exceptions or subclasses of those exceptions are allowed to be thrown;
- The methods must not be static. (If they are, the method is hidden and not overridden);
- @Override - It is a great idea to get in the habit of using it in order to avoid accidentally overloading a method.

Overloading precedence

- Exact match by type
- Matching a superclass type
- Converting to a larger primitive type
- Converting to an autoboxed type
- Varargs

Overloading precedence


```
class Overloading {
    static void overloadedMethod(int i) { // will enter if nothing is commented
        System.out.println("in int");
    }

    static void overloadedMethod(long i) { // will enter if comment 'int'
        System.out.println("in long");
    }

    static void overloadedMethod(Integer i) { // will enter if comment 'int' and 'long'
        System.out.println("in Integer");
    }

    static void overloadedMethod(Number i) { // will enter if comment 'int', 'long' and 'Integer'
        System.out.println("in Number");
    }

    static void overloadedMethod(int... i) { // will enter if comment 'int', 'long', 'Integer' and 'Number'
        System.out.println("in var arg");
    }

    public static void main(String[] args) {
        overloadedMethod(10);
    }
}
```

Interface

- Defines a set of public abstract methods, which classes implementing the interface must provide.
- Allows you to define what a class can do without saying how to do it (interface is a contract)
- A class may implement multiple interfaces as well as extend classes that implement interfaces, allowing for limited multiple inheritance in Java
- May extend other interfaces, although they may not extend a class and vice versa
- May contain public static final constant values, public and private methods, public default methods.

Interface

```
interface Walk {
    int someConst = 1;
    public static final int anotherConst = 1;

    boolean isQuadruped();

    private void doSomething() {
        //do some work
    }

    abstract double getMaxSpeed();
}

interface Run extends Walk {
    public abstract boolean canHuntWhileRunning();

    default double getMaxSpeed(){
        return 1;
    }
}
```

```
class Lion implements Run {
    public boolean isQuadruped() {
        return true;
    }

    public boolean canHuntWhileRunning() {
        return true;
    }

    public double getMaxSpeed() {
        return 100;
    }
}
```

Interface

Provides a way for one individual to develop code that uses another individual's code, without having access to the other individual's underlying implementation. Interfaces can facilitate rapid application development by enabling development teams to create applications in parallel, rather than being directly dependent on each other (for example one team uses interface, another team implements it)

Functional interface


```
@FunctionalInterface
interface Speakable {
    String say();

    static void someStatic(){}

    default void someDefault(){}

    private void doSomething(){}
}
```

Nested Classes

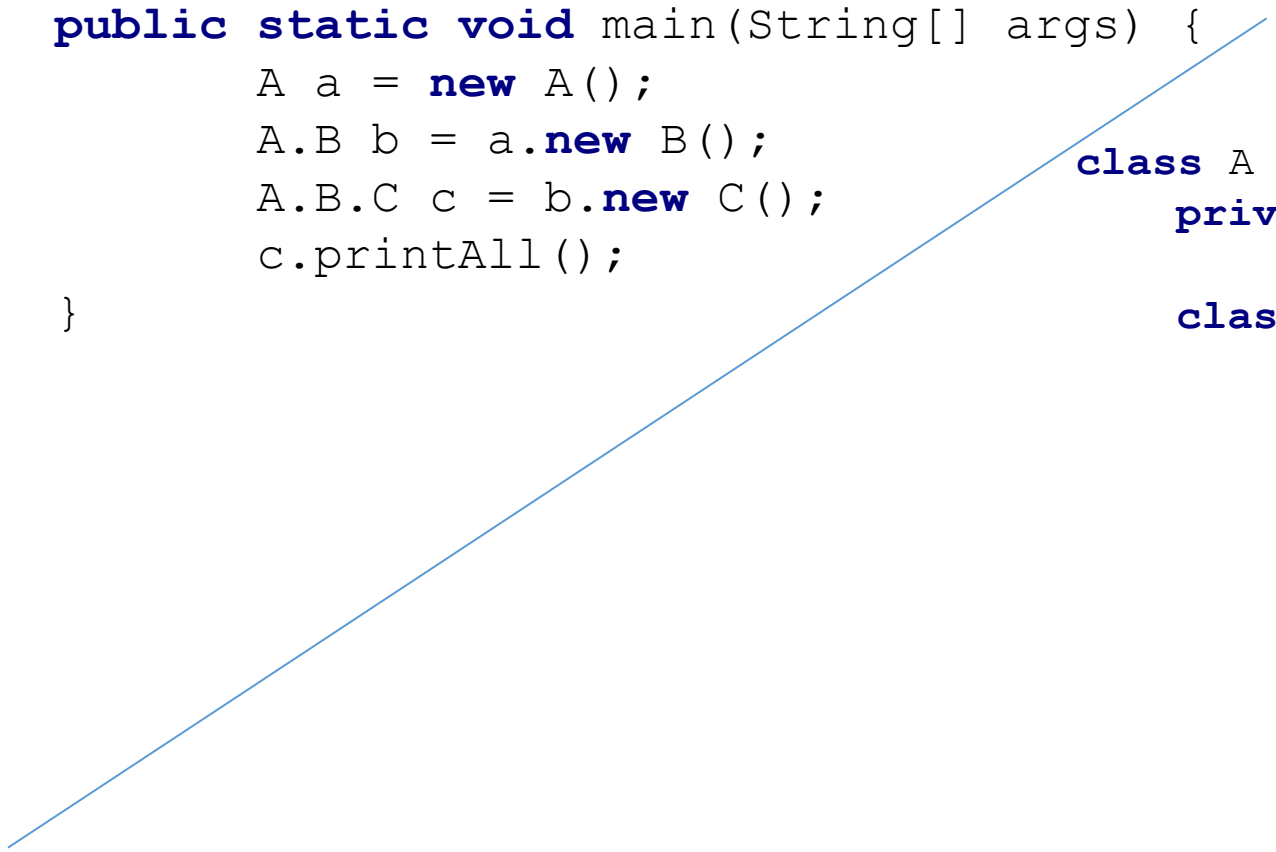
Types of nested classes:

- A member inner class is a class defined at the same level as instance variables. It is not static. Often, this is just referred to as an inner class without explicitly saying the type.
- A local inner class is defined within a method.
- An anonymous inner class is a special case of a local inner class that does not have a name.
- A static nested class is a static class that is defined at the same level as static variables.

Nested Classes

- encapsulate helper classes by restricting them to the containing class
- make it easy to create a class that will be used in only one place
- can make the code easier to read.

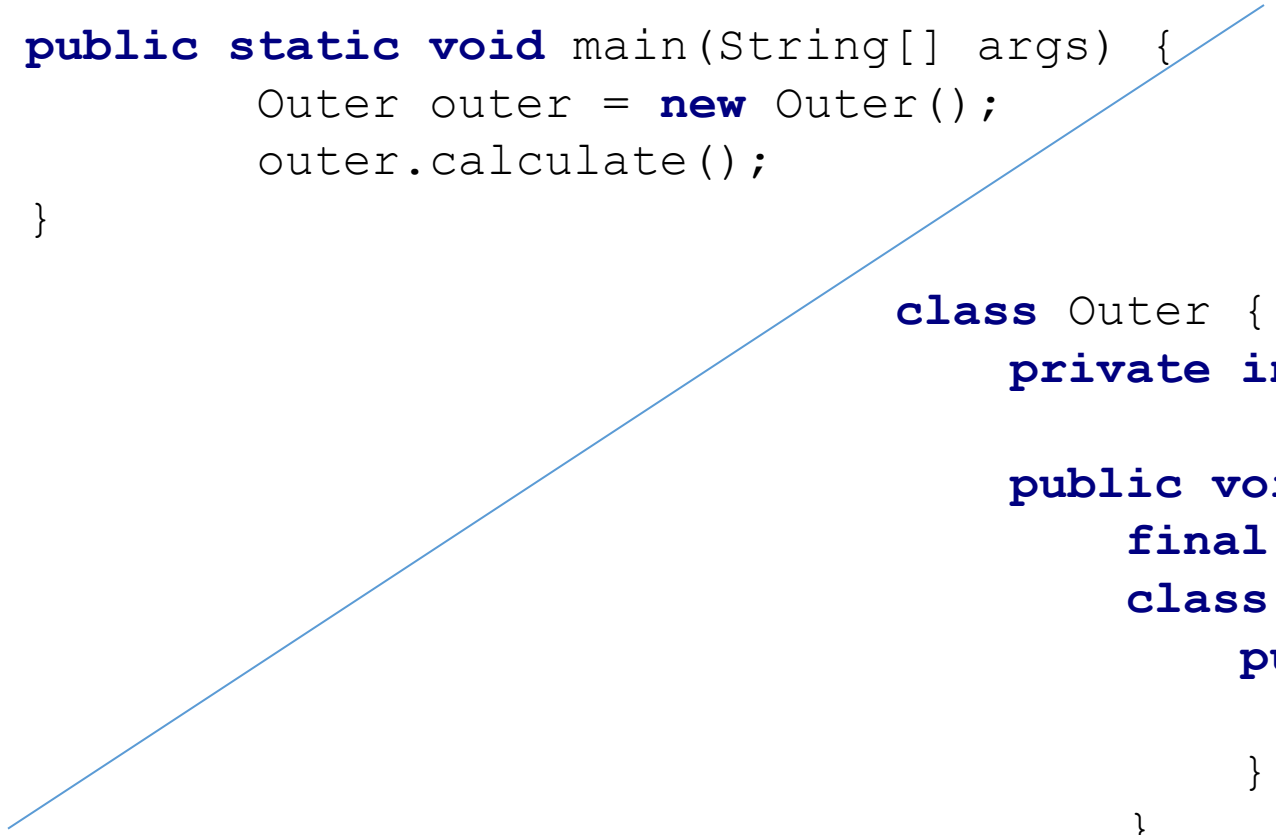
Member Inner Classes



```
public static void main(String[] args) {
        A a = new A();
        A.B b = a.new B();
        A.B.C c = b.new C();
        c.printAll();
}
```

```
class A {
    private int x = 1;

    class B {
        private int x = 2;

        class C {
            private int x = 3;

            public void printAll() {
                System.out.println(x); // 3
                System.out.println(this.x); // 3
                System.out.println(B.this.x); // 2
                System.out.println(A.this.x); // 1
            }
        }
    }
}
```

Local Inner Classes



```
public static void main(String[] args) {
        Outer outer = new Outer();
        outer.calculate();
}
```

```
class Outer {
    private int length = 5;

    public void calculate() {
        final int width = 20;
        class Inner {
            public void multiply() {
                System.out.println(length * width);
            }
        }
        Inner inner = new Inner();
        inner.multiply();
    }
}
```

Anonymous Inner Classes


```
public static void main(String[] args) throws Exception {
    JButton button = new JButton("red");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            // handle the button click
        }
    });
}
```

Static Nested Classes


```
class Enclosing {
    static class Nested {
        private int price = 6;
    }

    public static void main(String[] args) {
        Nested nested = new Nested();
        System.out.println(nested.price);
    }
}
```

Enum

- A special data type that enables for a variable to be a set of predefined constants
- Has unique set of values
- Can implement interface
- Can NOT extend class
- Can contain fields and methods
- Singleton

Enum

```
enum Planet implements IGravitable {
    MERCURY (3.303e+23, 2.4397e6) {
        @Override
        double surfaceGravity() {
            return -1;
        }
    },
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6)
    //    ...
    ;

    private double mass;   // in kilograms
    final double radius;   // in meters

    Planet(double mass, double radius) {// private
        this.mass = mass;
        this.radius = radius;
    }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    @Override
    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
}
```

```
static void main(String[] args) {
    double mass = Planet.EARTH.surfaceGravity();

    for (Planet p : Planet.values()){
        System.out.println("Your weight on %s is %f%n", p,
          p.surfaceWeight(mass));
    }

    System.out.println("Is Earth inhabited: " +
          isPlanetInhabited(Planet.EARTH));;
}

static boolean isPlanetInhabited(Planet alivePlanet) {
    switch (alivePlanet){
        case EARTH:
            return true;
        case VENUS:
        case MERCURY:
        default:
            return false;
    }
}
```

Design principle

- A design principle is an established idea or best practice that facilitates the software design process.

Design principle

- More logical code
- Code that is easier to understand
- Classes that are easier to reuse in other relationships and applications
- Code that is easier to maintain and that adapts more readily to changes in the application requirements

OOP principles

- Class
- Object
- Inheritance
- Encapsulation
- Polymorphism

OOP principles

Everything is object

Object is a class instance

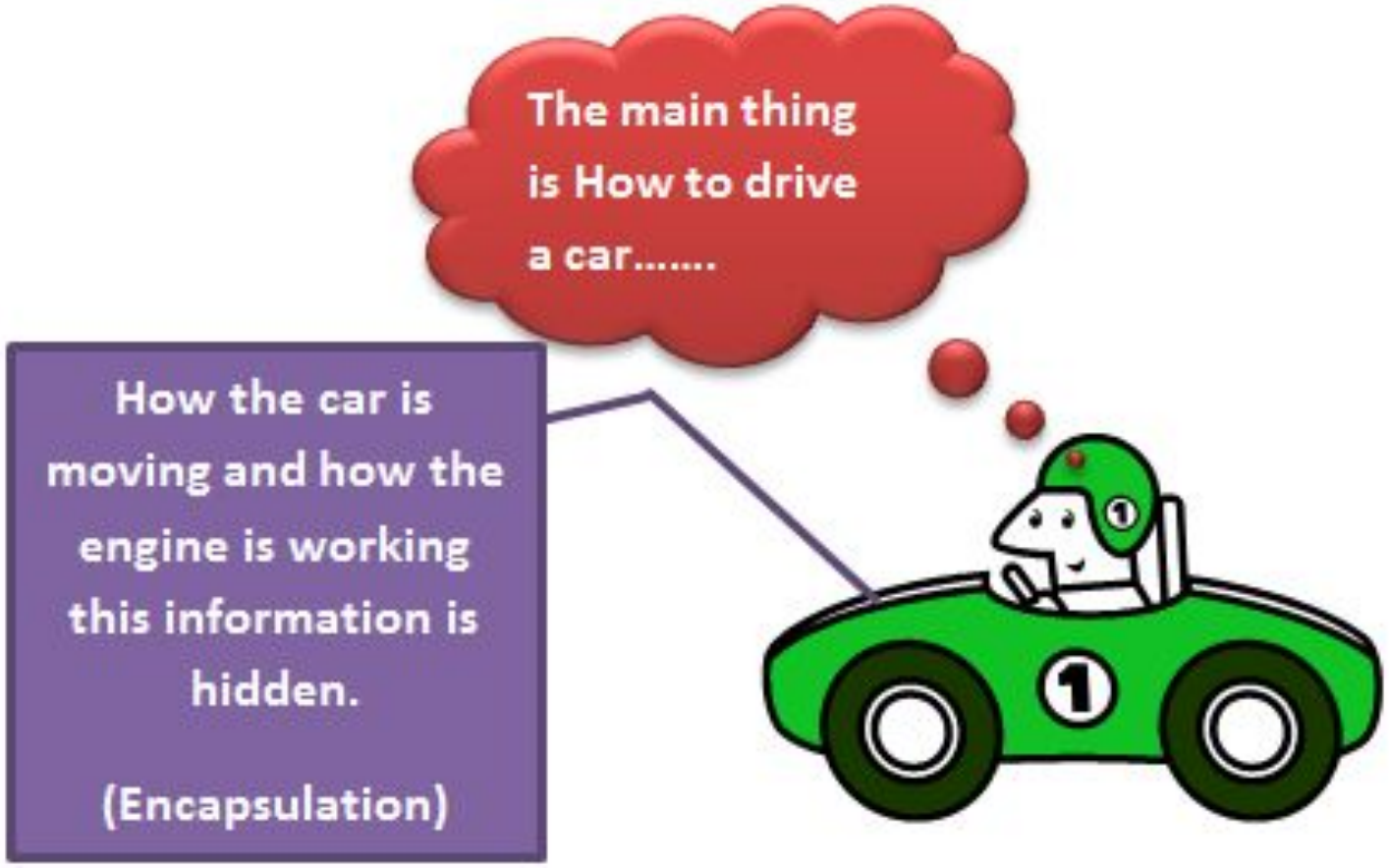
Program – a set of interacting objects

Object has a state and behavior

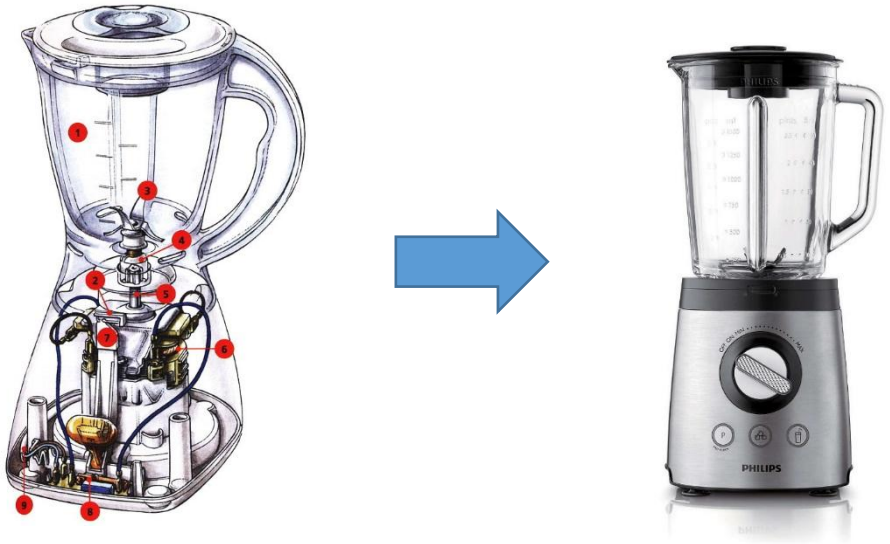
Access modifiers

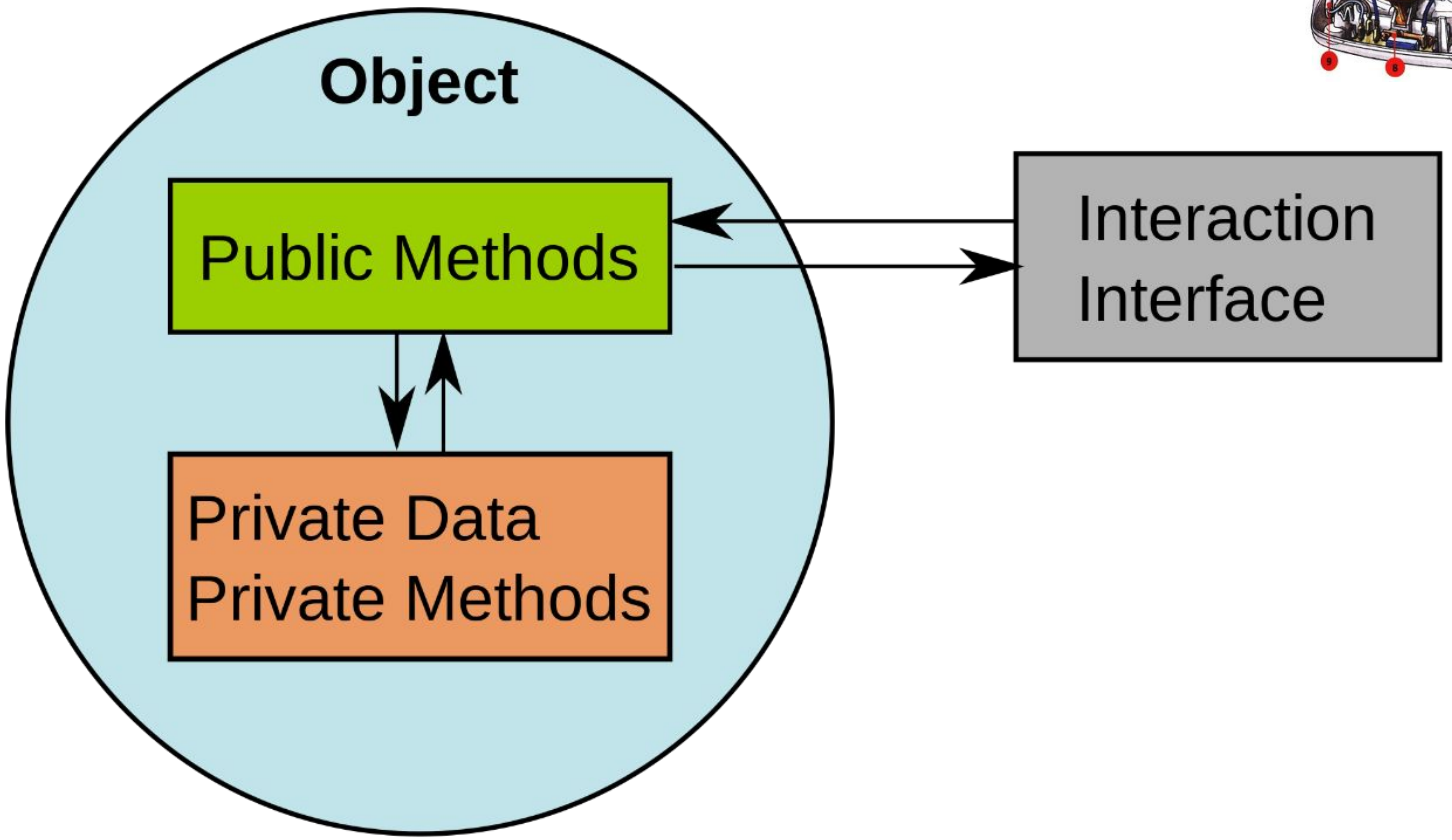

```
public class Dog {
    public String name = "Scooby-doo";
    protected boolean hasFur = true;
    boolean hasPaws = true;
    private int id;
}
```

Encapsulation



Encapsulation





Encapsulation

- No actor other than the class itself should have direct access to its data. The class is said to encapsulate the data it contains and prevent anyone from directly accessing it.

  With encapsulation, a class is able to maintain certain invariants about its internal data. An invariant is a property or truth that is maintained even after the data is modified.

Encasulation example

```
class CoffeeMachine {
    private int sugar;
    private double milk;

    Coffee makeCoffee(){
        blendBeans();
        heatWater();
        shakeMilk();
        return mix();
    }

    private Coffee mix() {
        return new Coffee(); // mix all together
    }

    private void shakeMilk() {
        // shaking milk
    }

    private void heatWater() {
        // heating water
    }

    private void blendBeans() {
        // blending beans
    }
}
```

```
public static void main(String[] args) {
    CoffeeMachine cm = new CoffeeMachine();
    Coffee coffee = cm.makeCoffee();
}
```

Encapsulation. Java Beans

A JavaBean is a design principle for encapsulating data in an object in Java.

- private properties
- public getter(get, is for primitive boolean)
- public setter (set)
- property name in getter/setter starts with uppercase

Encapsulation. Java Beans

What is wrong

```
class Girl {
    private boolean playing;
    private Boolean dancing;
    public String name;
    …
}
```

```
public boolean isPlaying() {
    return playing;
}

public boolean getPlaying() {
    return playing;
}

public Boolean isDancing() {
    return dancing;
}

public String name() {
    return name;
}

public void updateName(String n) {
    name = n;
}

public void setname(String n) {
    name = n;
}
```

Encapsulation. Java Beans

What is wrong

```
class Girl {
    private boolean playing;
    private Boolean dancing;
    public String name;
    …
}
```

```
public boolean isPlaying() {   // OK
    return playing;
}

public boolean getPlaying() {      // OK
    return playing;
}

public Boolean isDancing() {   // BAD, get for wrapper
    return dancing;
}

public String name() {    // BAD, not get prefix
    return name;
}

public void updateName(String n) {  // BAD, no set prefix
    name = n;
}

public void setname(String n) { // BAD, wrong case
    name = n;
}
```

Polymorphism

An object in Java may take on a variety of forms, in part depending on the reference used to access the object.

One name, many forms. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality.

There are 2 basic types of polymorphism.

- Overriding, also called run-time (dynamic) polymorphism.
- Overloading, which is referred to as compile-time (static) polymorphism.

Polymorphism

The type of the object determines which properties exist within the object in memory.

The type of the reference to the object determines which methods and variables are accessible to the Java program

Polymorphism

Polymorphism is the ability of a single interface to support multiple underlying forms.

```
interface LivesInOcean {
    void makeSound();
}

class Dolphin implements LivesInOcean {
    public void makeSound() {
        System.out.println("whistle");
    }
}

class Whale implements LivesInOcean {
    public void makeSound() {
        System.out.println("sing");
    }
}
```

```
class Oceanographer {
    void checkSound(LivesInOcean animal) {
        animal.makeSound();
    }

    void main(String[] args) {
        Oceanographer o = new Oceanographer();
        o.checkSound(new Dolphin());
        o.checkSound(new Whale());
    }
}
```

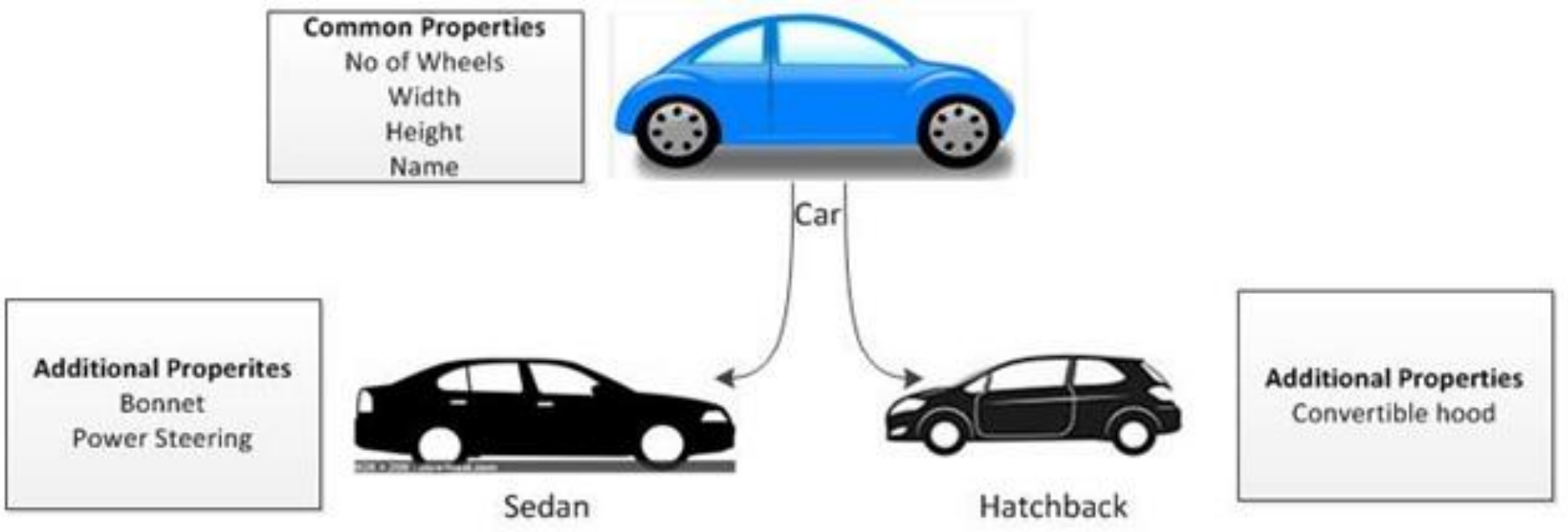
Polymorphism

Polymorphism also allows one object to take on many different forms.

```
class Primate {
    public boolean hasHair() {
        return true;
    }
}

interface HasTail {
    boolean isTailStriped();
}

class Lemur extends Primate implements HasTail
{
    public int age = 10;

    public boolean isTailStriped() {
        return false;
    }
}
```

```
public static void main(String[] args) {
    Lemur lemur = new Lemur();
    println(lemur.age);

    HasTail hasTail = lemur;
    println(hasTail.isTailStriped());

    Primate primate = lemur;
    println(primate.hasHair());
}
```

Inheritance (is-a)



Inheritance (is-a)

```
abstract class Animal {
    String name;

    @Override
    public String toString() {
        return name;
    }

    abstract void feed();
}
```

```
class Cow extends Animal {
    @Override
    public void feed() {
    }

    public void run(){
    }
}
```

```
class Bird extends Animal {
    @Override
    public void feed() {
    }

    public void fly(){
    }
}
```

Virtual methods invocation

```
abstract class Animal {
    public abstract void feed();
}
```

```
public void feedAnimal(Animal animal) {
    animal.feed();
}
```

```
class Cow extends Animal {
    public void feed() {
        addHay();
    }

    private void addHay() {
    }
}

class Bird extends Animal {
    public void feed() {
        addSeed();
    }

    private void addSeed() {
    }
}

class Lion extends Animal {
    public void feed() {
        addMeat();
    }

    private void addMeat() {
    }
}
```

Composition (has-a)

Object composition is the idea of creating a class by connecting other classes as members using the has-a principle.

Inheritance is the idea of creating a class that inherits all of its reusable methods and objects from a parent class.

Both are used to create complex data models, each with its own advantages and disadvantages

Composition (has-a)


```
class Person {
    Job job;

    public Person() {
        this.job = new Job();
        job.setSalary(1000L);
    }

    public long getSalary() {
        return job.getSalary();
    }

}
```

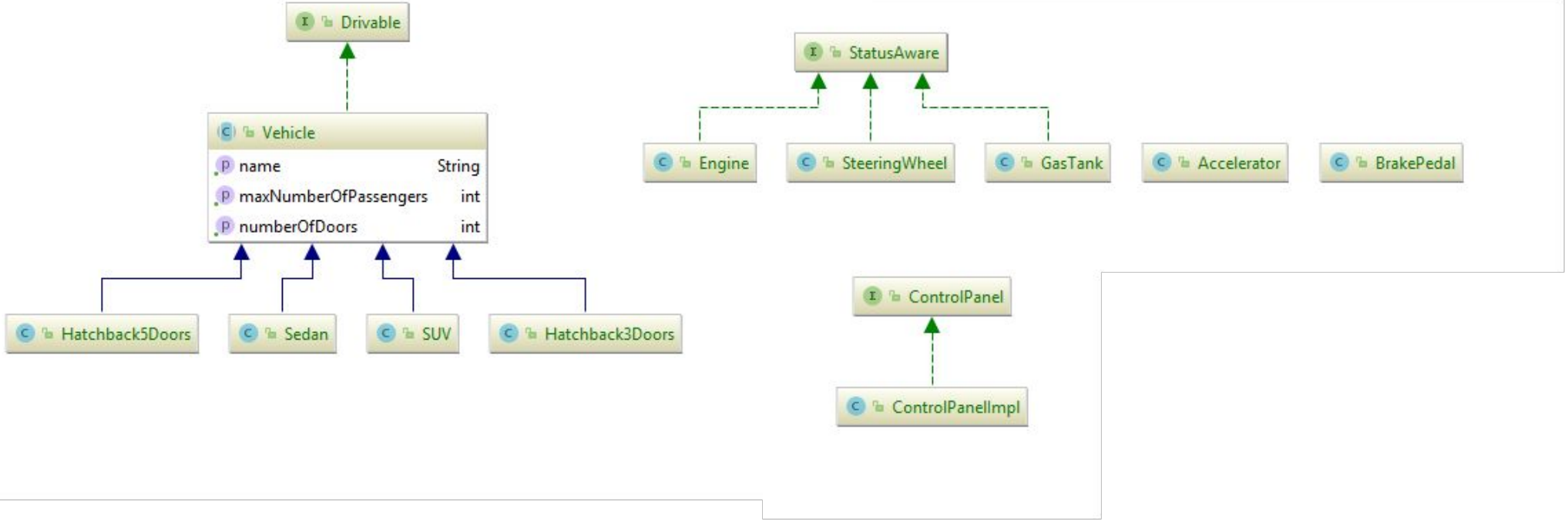
```
class Job {
    private long salary;

    public void setSalary(long salary) {
        this.salary = salary;
    }

    public long getSalary() {
        return salary;
    }
}
```

Literature

- https://docs.oracle.com/javase/tutorial/java/javaOO/index.html
- https://docs.oracle.com/javase/tutorial/java/IandI/index.html
- https://docs.oracle.com/javase/tutorial/java/concepts/index.html

Homework

- Implement classes: SUV, Sedan, Hatchback3Doors, Hatchback5Doors
- SUV, Sedan, Hatchback3Doors, Hatchback5Doors should extend Vehicle
- Vehicle provides info: name, max passengers number, number of doors
- Vehicle should implement Drivable
- Each Vehicle contains Accelerator, BrakePedal, Engine, GasTank, SteeringWheel
- Implement Accelerator, BrakePedal, Engine, GasTank, SteeringWheel according to their names:
    - Accelerator works 5 seconds (5 times) and speed-up the car for a some **accelerateStrength**
    - BrakePedal slow down the car for a some **brakingStrength**
    - Engine hat it **capacity, max speed,** can be **started** or **stopped**, uses fuel from GasTank when work
    - GasTank has it **max** and **current volumes,** GasTank can use fuel or can be filled by it
    - SteeringWheel has it **max turn angle, current turn angle** and **step** (one turn changes the current angle for this value), SteeringWheel can be turned left or right
- Verify all properties of each car part they cannot be greater than some max value and less than some min value (see tests); speed, volume cannot be negative or greater than max speed and/or volume
- Engine, GasTank, SteeringWheel should implement interface StatusAware
- Create class ControlPanel that must control the Drivable regardless of what type of vehicle it is transmitted


Drivable
Vehicle
name String
maxNumberOfPassengers int
numberOfDoors int
Hatchback5Doors
Sedan
SUV
Hatchback3Doors
StatusAware
Engine
SteeringWheel
GasTank
Accelerator
BrakePedal
ControlPanel
ControlPanelImpl