

Inheritance

Best practises



IT Learning &
Outsourcing Center

www.pragmatic.bg

Lector: Stefan Vadev

Skype: stefanvadev77

E-mail: stefan.vadev@gmail.com

Facebook:

<https://www.facebook.com/stefan.vadev.7>

Copyright © Pragmatic LLC

2013–2018



Table of contents

- Basics – what is inheritance, when to use it , why use it
- Class – The basic unit of a java program. What's a package, static field and access modifiers
- Constructors in the context of inheritance
- Method overriding
- Examples from the Java API. Class Object



Basics

- Inheritance is one of principles of the OOP
- Inheritance is the ability of a class to implicitly gain some(or all) members of another class
- Inheritance builds a relationship between classes
- The class that gains the functionality is called subclass (child class or derived class). The other class is called base(parent or superclass)
- The main purpose of inheritance is reusability of the code

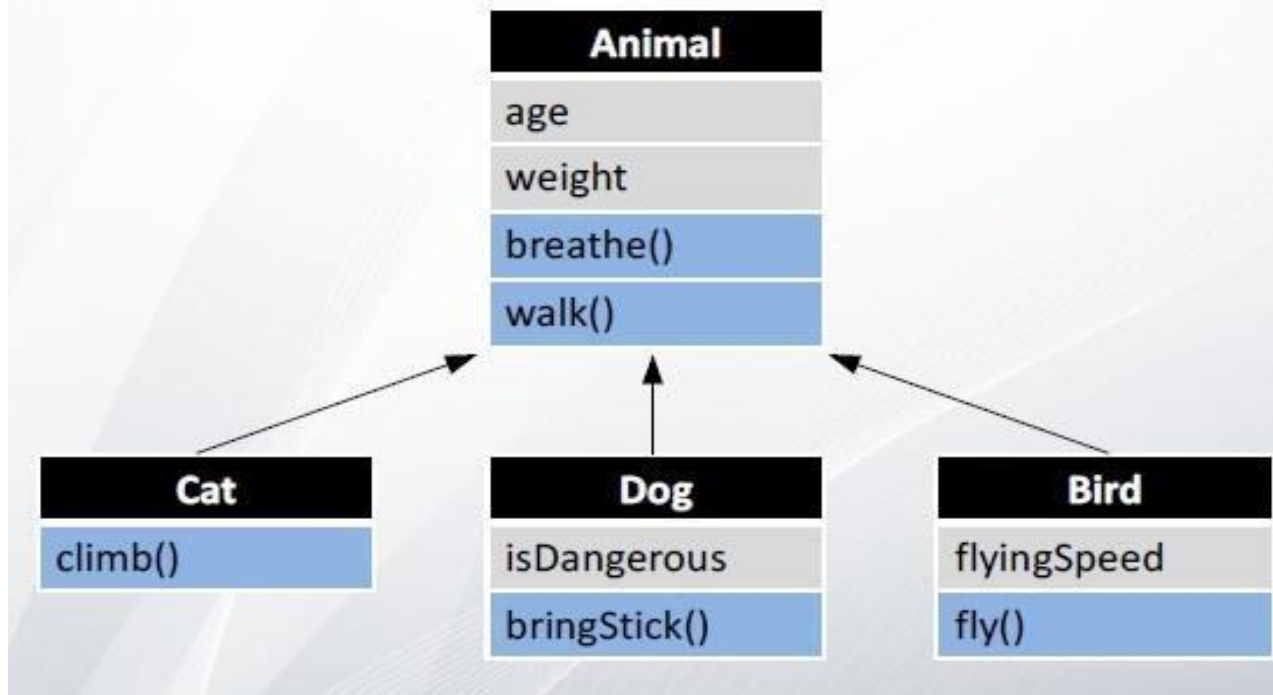


Problem

| Cat | Dog | Bird |
|-----------|--------------|-------------|
| age | age | age |
| weight | weight | weight |
| breathe() | isDangerous | flyingSpeed |
| walk() | breathe() | breathe() |
| climb() | walk() | walk() |
| | bringStick() | Fly() |

- Cats, dogs and birds have age and weight.
- They also can breathe and walk.
- Actually, they are animals and every animal has age, weight and can breathe and walk.

Problem



- Is this the same?
- Yes, but the common logic is only in the class **Animal**



Inheritance syntax in Java

- Inheritance in java is provided through the keyword *extends* i.e the subclass class extends the superior
- Each class can extend only one class



Animal example

```
package com.pragmatic.lesson5.inheritance.simple

public class Animal {
    int age;

    double weight;

    void breathe() {
        System.out.println("Breathing...");
    }
    void walk() {
        System.out.println("Walking...");
    }
}
```



Animal example

```
public class Cat extends Animal{  
    void climb() {  
        System.out.println("Climbing...");  
    }  
}
```

Keyword
extends

```
public class Dog extends Animal{  
    boolean isDangerous;  
  
    void bringStick() {  
        System.out.println("Bringing the stick...");  
    }  
}
```

```
public class Bird extends Animal{  
    double flyingSpeed;  
  
    void fly() {  
        System.out.println("Flying...");  
    }  
}
```




'is a' and a 'has a' relationship

- Two of the main techniques for code reuse are class inheritance and object composition
- 'is a' relationship is expressed with inheritance
- 'has a' relationship is expressed with composition
- Example using classes House, Building and Bathroom
- Is a: House is a Building
- has a: House has a Bathroom



Inheritance vs Composition

- Inheritance is uni-directional:
- House is a Building. But Building is not a House.
- Inheritance uses *extends* key word

```
class Building {  
    //...  
}  
  
class House extends Building {  
    //...  
}
```



Keyword
extends



- Composition is used when House has a Bathroom. It is incorrect to say House is a Bathroom
- Composition simply means using instance variables that refer to other objects. The class House will have an instance which refers to a Bathroom object

```
class House {  
    Bathroom bathroom;  
    // ...  
}
```



Inheritance Example

- Now lets take a look everything we've said about inheritance under the `com.pragmatic.lesson5.inheritance.simple` package



The Class - Package

- A Java package is a mechanism for organizing Java classes into namespaces
- Hierarchical units identical to folders – on the file system the packages are presented as folders
- Provide grouping of related types(classes)
- Provide access protection and space management
- Package names cannot contain java key words



Method overriding

- A subclass can predefine the methods of its parent
- This is called overriding
- To override a method:
 - Use its name as it's in the parent class
 - Use the same number and order of method arguments
 - Use the same return type(or subtype of the type returned by the overridden method. This is called a covariant return type)



Method overriding example

- Birds can fly. The eagle and the sparrow are birds and can fly. Although the eagle can fly highly and fast.

```
public class Eagle extends Bird{  
    public void fly(){  
        System.out.println("Flying highly like an eagle");  
    }  
}
```

```
public class Sparrow extends Bird{  
    public void fly(){  
        System.out.println("Flying like a sparrow");  
    }  
}
```



More about overriding

- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared private are not inherited at all, so there is no rule for them.



More about overriding

- Overriding is used for Polymorphism
- (we'll talk about this in the next lesson)
- You can invoke the parent method when overriding it - keyword *super* is used
- Try to override method `startEngine` in class `SportCar`.
- In the body of the method first invoke `startEngine` of the parent, and then switch on the turbo.



More about overriding

```
public class Car {  
    ...  
    public void startEngine() {  
        System.out.println("Starting engine...");  
    }  
    ...  
}
```

Overriding
startEngine()

```
public class SportCar extends Car {  
    ...  
    public void startEngine() {  
        super.startEngine();  
        switchTurbo(true);  
    }  
}
```

Invoking
startEngine() of
parent class

Try to override equals(Object obj) and toString() too.



Class - Access modifiers

- Access modifiers are used to
 - Control access to classes (top level), methods, constructors or fields (bottom level) from outside the class
- For top level (classes) there are *public*, *package* and in some cases *private*(inner classes)
- For bottom level: *public*, *protected*, *package* and *private*

Explaining *public*, *private* and *default*



- *public* – gives access to the class, field or method from everywhere outside the class
- *private* – access is restricted only within the class
- *default/package* – visible from within the class and all other classes in the package
- *protected* – the fields and methods are visible to the *classes* within the same package and in the child classes
- Nice explanation - <http://www.java-made-easy.com/java-access-modifiers.html>

Access modifiers in inheritance



- *public*, *private* and default(package)
- *protected* fields and methods is visible to the *classes* within the same package and in the child classes
 - Table with access levels:

| | Class | Package | Subclass (in the same pkg) | Subclass | World |
|-----------|-------|---------|-------------------------------|----------|-------|
| public | Yes | Yes | Yes | Yes | Yes |
| protected | Yes | Yes | Yes | Yes | No |
| default | Yes | Yes | Yes | No | No |
| private | Yes | No | No | No | No |



Purpose of accessmodifiers

- Problem: If all fields of class Person are public they will be accessible from everywhere which violates the Encapsulation principle of OOP
- Accessibility directly to fields is dangerous and unsecure
- For accessing private fields outside the class are used public methods called „getter“ and „setter“
- Lets demonstrate some things under the `com.pragmatic.lesson5.inheritance.cars` package



Getters and setters

- Getters are used for getting the value of private field outside the class.
- It should be implemented only if is necessary
- Setters are void methods and are used for setting the value of private field outside the class
- Validation can be implemented as part of the setter's body



Getter and setter example

```
private int age;

public int getAge() {
    return age;
}

public void setAge(int age) {
    if (age >= 0) {
        this.age = age;
    }
}
```




Keyword *final*

- **Final** is a keyword or reserved word in java and can be applied to member variables, methods, class and local variables in Java.
- You use the **final** keyword in a **method** declaration to indicate that the method cannot be overridden by subclasses
- A **class** that is declared **final** cannot be inherited
- Setting **fields** or **argument** of some reference type as **final** don't guarantee that its state won't be changed. It only guarantee that the reference won't be changed. In other words, **final** is only about the reference itself, and not about the contents of the referenced object.



Using keyword *final* for method's parameters

- The same logic as when using with fields – the parameter cannot be changed in the method's body

```
public void setAgeFromOtherPerson(final Person person)
{
    this.age = person.getAge();
}
```

- !!! Be careful with fields and parameters of some reference type. Once again:
- Setting fields or argument of some reference type as final don't guarantee that its state won't be changed. It only guarantee that the reference won't be changed.



Using keyword *final* for variable in some block of code

```
public class Demo {  
    public static void main(String[] args) {  
        Car bmw = new Car("BMW 330", true, "Red");  
        Car ford = new Car("Ford Fiesta", false,  
"Black", 2000, 330);  
        final Car MY_CAR = bmw;  
        MY_CAR = ford; // !!!COMPILE ERROR!!!  
  
        final int MY_AGE = 20;  
        MY_AGE = 21; // !!!COMPILE ERROR!!!  
    }  
}
```



Static fields

- Keyword *static* indicate the field as static
- Static fields belong to the class – not the instances of a class
- Static fields are shared between the objects because they belong to the class
- Static reference can be and should be referenced via class' name, example: `Class.field`



Static fields

- If some object change the value of a static fields, its changed in all object of this class
- Try it with few simple objects!



Subclass Constructors

- Invocation of a superclass constructor must be the first line in the subclass constructor
- Keyword *super* is used for this
- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the default constructor of the superclass.

Constructor chaining –Part 1



- When a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, there will be a whole chain of constructors calls

```
public class A {  
    public A() {  
        System.out.println("Calling constructor of class A");  
    }  
}  
  
public class B extends A {  
    public B() {  
        System.out.println("Calling constructor of class B");  
    }  
}  
  
public class C extends B {  
    public C() {  
        System.out.println("Calling constructor of class C");  
    }  
}
```

Constructor chaining –Part 2



Console X

```
<terminated> ChainTest [Java Application] C:\Program Files\Java\
```

```
Calling constructor of class A
```

```
Calling constructor of class B
```

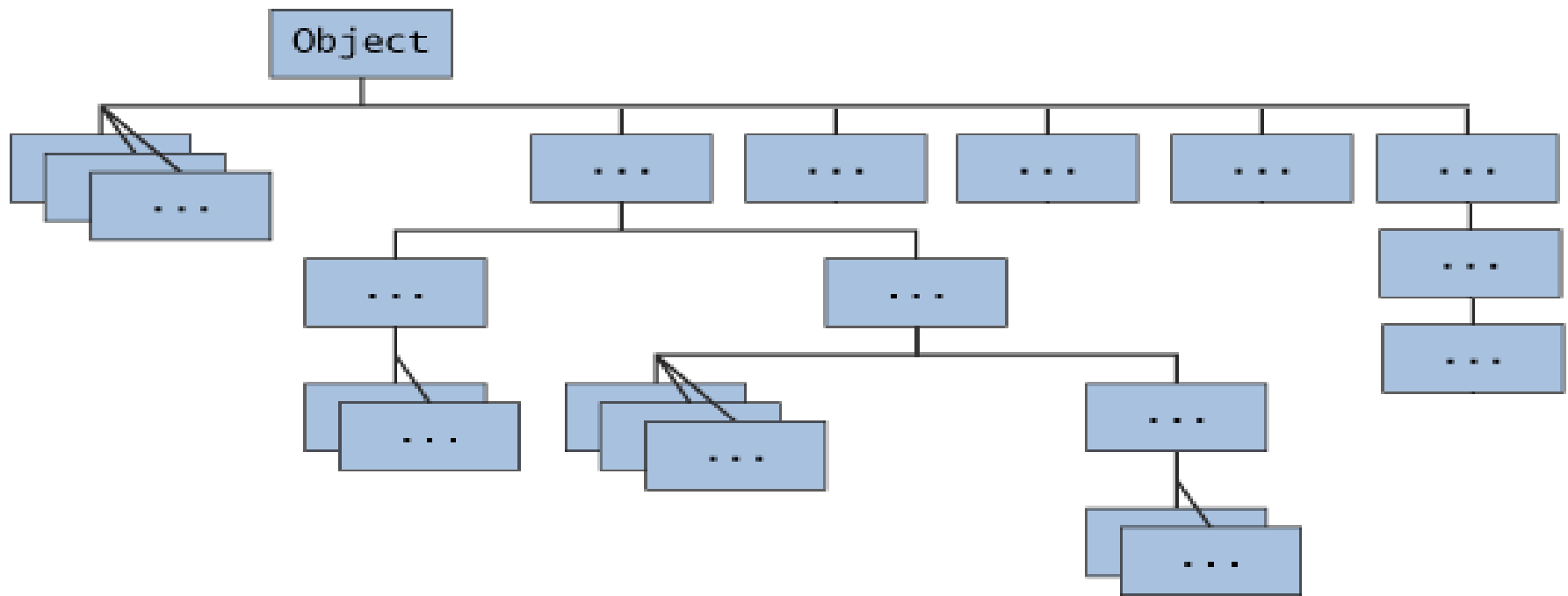
```
Calling constructor of class C
```

Now lets have a look at the example under
com.pragmatic.lesson5.inheritance.constructors



Java Class Hierarchy

- `java.lang.Object` is parent of all classes in Java
- It defines behavior common to all classes
- When you write class which don't extends other class, it implicitly derive from `Object`





Some methods of class Object

- Create a new class. Then instantiate it and use
- Ctrl + Space in eclipse to view Object's methods
- `boolean equals(Object obj)`
- Indicates whether some other object is "equal to" this one.
- `String toString()`
- Returns a string representation of the object.
- *(later we'll write some equals and toString methods)*

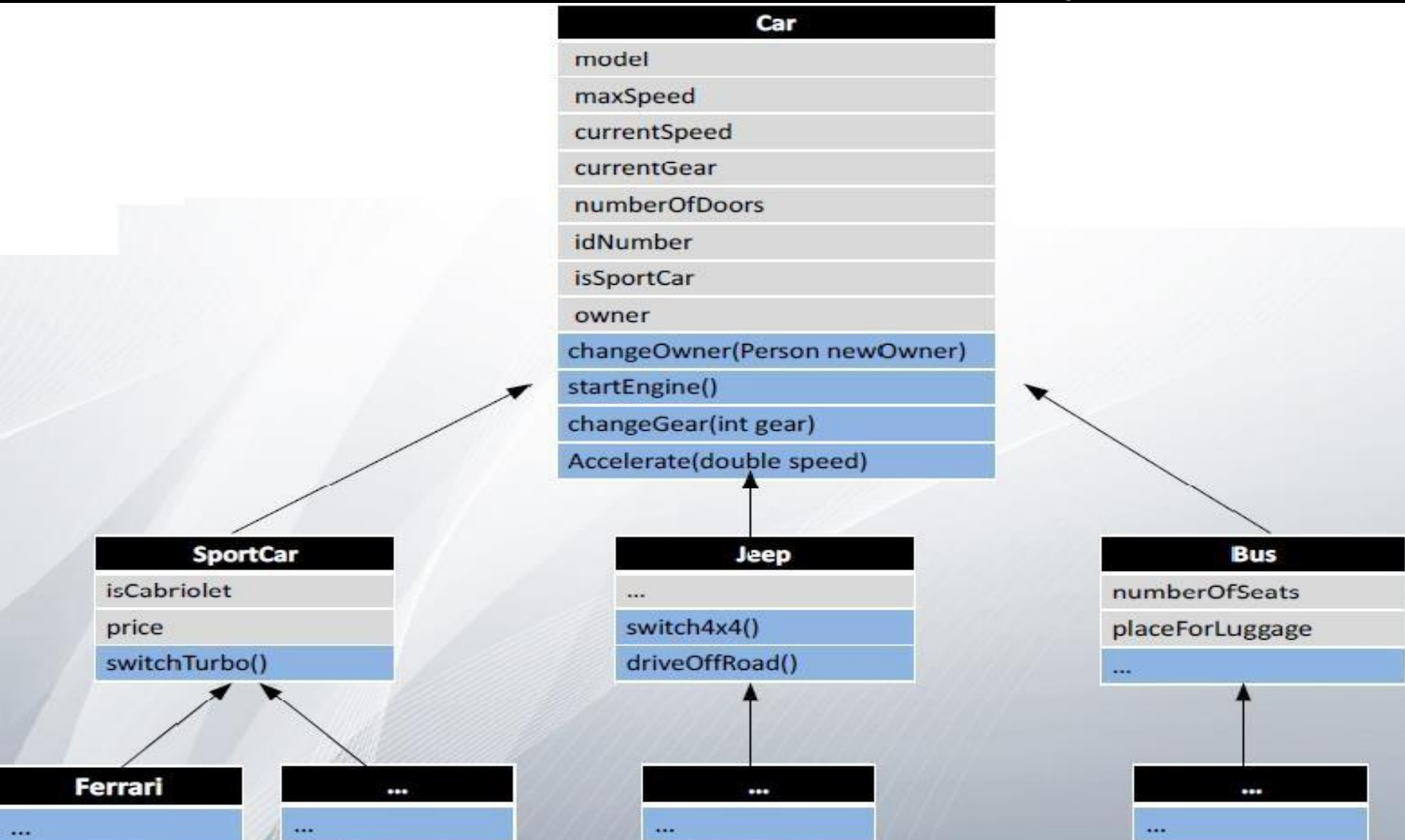


Cars example

- Let's create Car hierarchy to demonstrate all described in the previous slides
- We can use classes Car and Person from `com.pragmatic.lesson5.inheritance.cars`



Cars example





Using keyword *super* for fields and methods

- Subclasses classes can use functionality of the parent class through the keyword *super*
- Super can be used for fields, methods or constructors of the super class
- You can declare a field in the subclass with the same name as the one in the superclass (not recommended)
- Try setting `isSportCar` to `true` in class `Car`.
- What happen if class `SportCar` declare its own field `isSportCar`?



Summary

- What is inheritance and how to use it?
- 'Is a' and 'has a' relationship
- Access modifiers in context of inheritance
- Invoking constructors of the superclass and constructor chaining
- Java hierarchy
- How to use *super* keyword?
- What is method overriding?
- What is the constraints when overriding method?