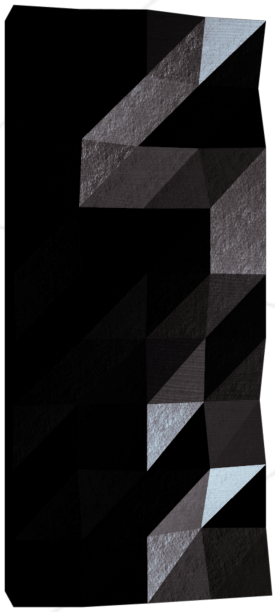


Object-oriented Programming

OOP Basics

Your Name mpas
42 Staff pedago@42.fr

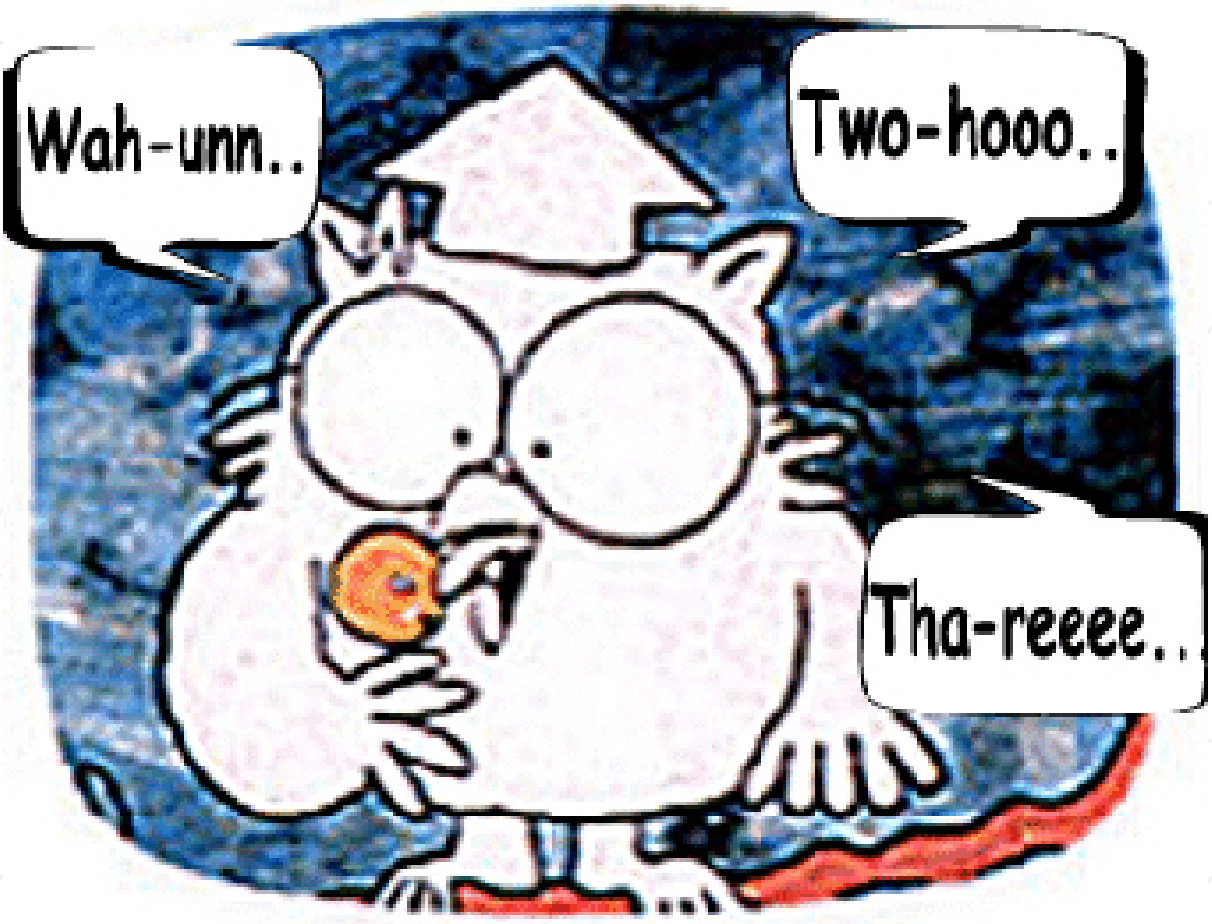
Summary: This document is an introduction to `Object-oriented Programming`

Contents

Chapter I

Foreword

A school of higher learning actually measured Tootsie Pop licks. It officially takes 364 licks to get to the center of a Tootsie Pop. Well, at least according to engineering students at Purdue University, who used a proprietary "licking machine" rather than a human tongue.



Chapter II

Introduction

Java is a class-based object-oriented programming (OOP) language that is built around the concept of objects. OOP concepts (OOP) intend to improve code readability and reusability by defining how to structure a Java program efficiently. The four principles of object-oriented programming are:

- abstraction
- encapsulation
- inheritance
- polymorphism

II.1 Abstraction

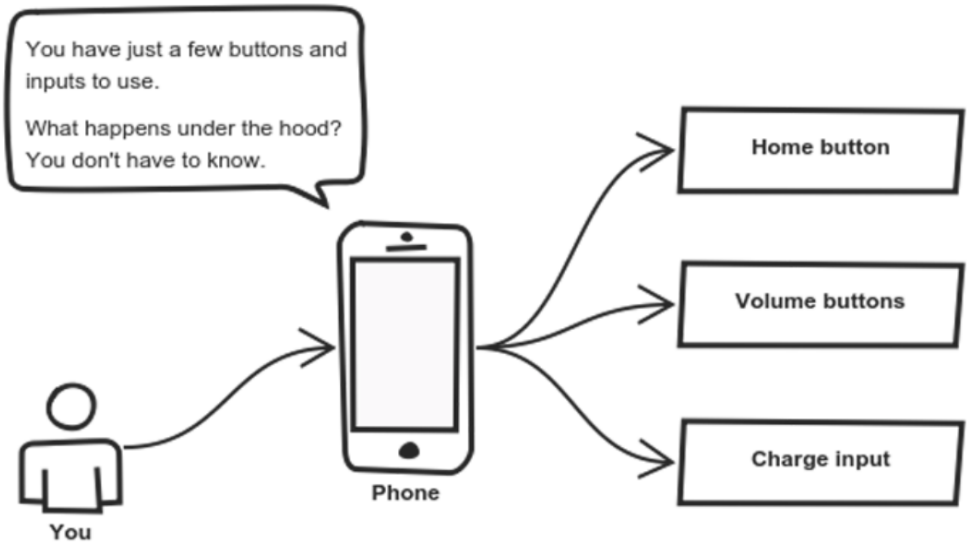
Abstraction aims to hide complexity from the users and show them only the relevant information. For example, if you want to drive a car, you don't need to know about its internal workings. The same is true of Java classes. You can hide internal implementation details by using abstract classes or interfaces. On the abstract level, you only need to define the method signatures (name and parameter list) and let each class implement them in their own way.

II.1.1 Abstraction in Java:

1. Hides the underlying complexity of data
2. Helps avoid repetitive code
3. Presents only the signature of internal functionality
4. Gives flexibility to programmers to change the implementation of the abstract behaviour
5. Partial abstraction (0-100%) can be achieved with abstract classes
6. Total abstraction (100%) can be achieved with interfaces

In object-oriented design, programs are often extremely large. And separate objects communicate with each other a lot. So maintaining a large codebase like this for years with changes along the way is difficult. **Abstraction is a concept aiming to ease this problem.** Applying abstraction means that each object should only expose a high-level mechanism for using it. This mechanism should hide internal implementation details. It should only reveal operations relevant for the other objects.
Think a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button.

Preferably, this mechanism should be easy to use and should rarely change over time. Think of it as a small set of public methods which any other class can call without "knowing" how they work. Another real-life example of abstraction? Think about how you use your phone:



Cell phones are complex. But using them is simple.

You interact with your phone by using only a few buttons. What's going on under the hood? You do not have to know implementation details are hidden. You only need to know a short set of actions. Implementation changes for example, a software update rarely affect the abstraction you use.

Check more about abstraction
https://www.geeksforgeeks.org/abstraction-in-java-2/

in Java, a separate keyword abstract is used to make a class abstract.

An example abstract class in Java ,by the way functions inside of the abstract class can be abstract or not.

II.1.2 Java code snippet

```
abstract class Shape {
    int color;
    abstract void draw();
}

abstract class Shape {
    int color;
  void draw();
}
```

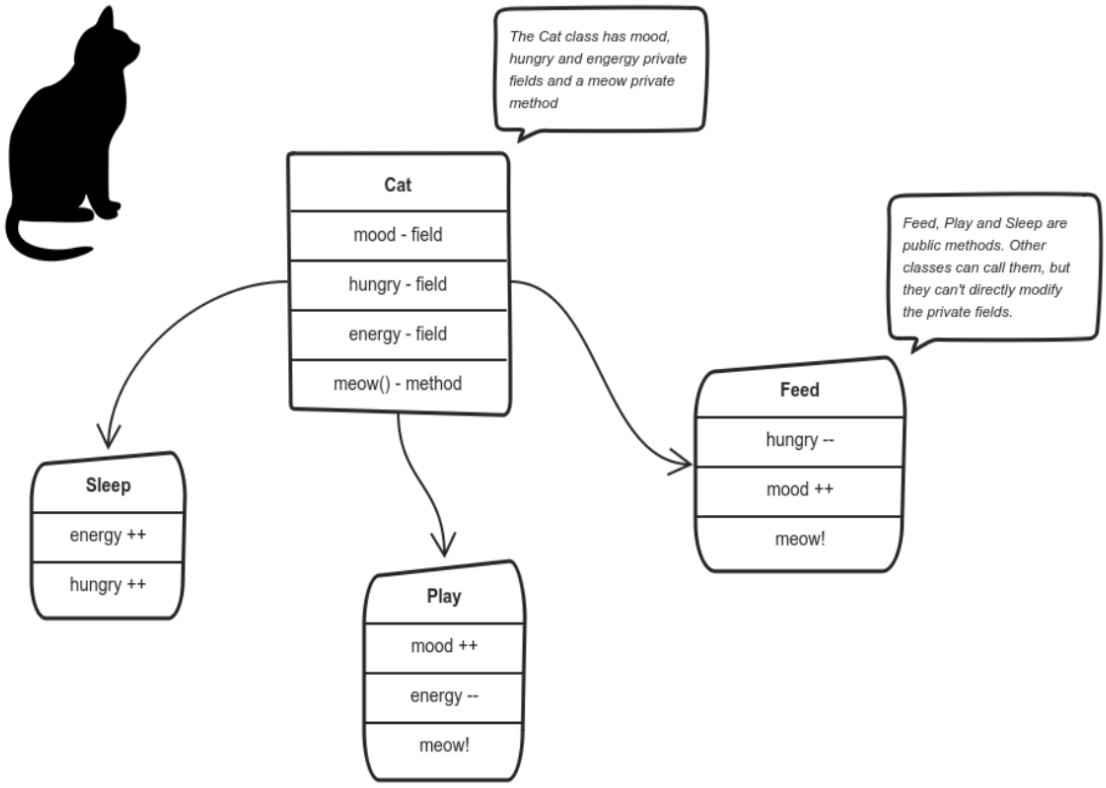
II.2 Encapsulation

Say we have a program. It has a few logically different objects which communicate with each other according to the rules defined in the program.

Encapsulation is achieved when each object keeps its state private, inside a class. Other objects don't have direct access to this state.Instead, they can only call a list of public functions called methods.

So, the object manages its own state via methods and no other class can touch it unless explicitly allowed. If you want to communicate with the object, you should use the methods provided. But (by default), you can't change the state.

Let's say we're building a tiny Sims game. There are people and there is a cat. They communicate with each other.We want to apply encapsulation, so we encapsulate all "cat" logic into a Cat class. It may look like this:



Here the "state" of the cat is the private variables mood, hungry and energy. It also has a private method meow(). It can call it whenever it wants, the other classes can't tell the cat when to meow. What they can do is defined in the public methods sleep(), play() and feed(). Each of them modifies the internal state somehow and may invoke meow(). Thus, the binding between the private state and public methods is made.**So encapsulation allows us to protect the data stored in a class from system-wide access.** As its name suggests, it safeguards the internal contents of a class like a real-life capsule. You can implement encapsulation in Java by keeping the fields (class variables) private and providing public getter and setter methods to each of them. Java Beans are examples of fully encapsulated classes.

II.2.1 Encapsulation in Java:

1. Restricts direct access to data members (fields) of a class.
2. Fields are set to private
3. Each field has a getter and setter method
4. Getter methods return the field
5. Setter methods let us change the value of of the field

Check more about encapsulation
https://www.geeksforgeeks.org/encapsulation-in-java/

Following is an example that demonstrates how to achieve Encapsulation in Java :

II.2.2 Java code snippet

```java
public class EncapTest {
   private String name;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }


   public void setAge( int newAge) {
      age = newAge;
   }

   public void setName(String newName) {
      name = newName;
   }
}

public class RunEncap {
   public static void main(String args[]) {
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);

      System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge());
      }
    }
```

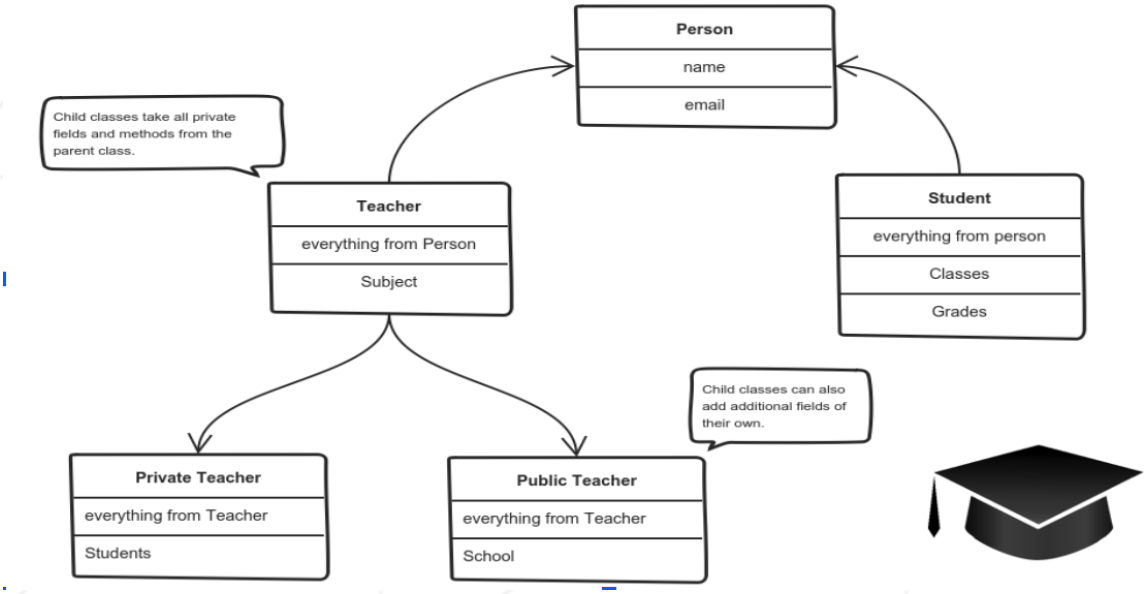
```
Name : James Age : 20/
```

II.3 Inheritance

OK, we saw how encapsulation and abstraction can help us develop and maintain a big codebase. But do you know what is another common problem in OOP design?

Objects are often very similar. They share common logic. But they're not entirely the same. Ugh... So how do we reuse the common logic and extract the unique logic into a separate class? One way to achieve this is inheritance.

It means that you create a (child) class by deriving from another (parent) class. This way, we form a hierarchy. The child class reuses all fields and methods of the parent class (common part) and can implement its own (unique part). For example:



(A private teacher is a type of teacher. And any teacher is a type of Person.)

If our program needs to manage public and private teachers, but also other types of people like students, we can implement this class hierarchy. This way, each class adds only what is necessary for it while reusing common logic with the parent classes. **So inheritance makes it possible to create a child class that inherits the fields and methods of the parent class. The child class can override the values and methods of the parent class, however it's not necessary. It can also add new data and functionality to its parent. Parent classes are also called superclasses or base classes, while child classes are known as subclasses or derived classes as well. Java uses the extends keyword to implement the principle of inheritance in code.**

II.3.1 Inheritance in Java:

1. A class (child class) can extend another class (parent class) by inheriting its features.
2. Implements the DRY (Don't Repeat Yourself) programming principle.
3. Improves code reusability.
4. Multilevel inheritance is allowed in Java (a child class can have its own child class as well).
5. Multiple inheritances are not allowed in Java (a class can't extend more than one class).

II.3.2 Java code snippet

```java
import java.util.*;
import java.lang.*;
import java.io.*;

class one
{
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class two extends one
{
    public void print_for()
    {
        System.out.println("for");
    }
}
// Driver class
public class Main
{
    public static void main(String[] args)
    {
        two g = new two();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

```
Geeks
for
Geeks
```

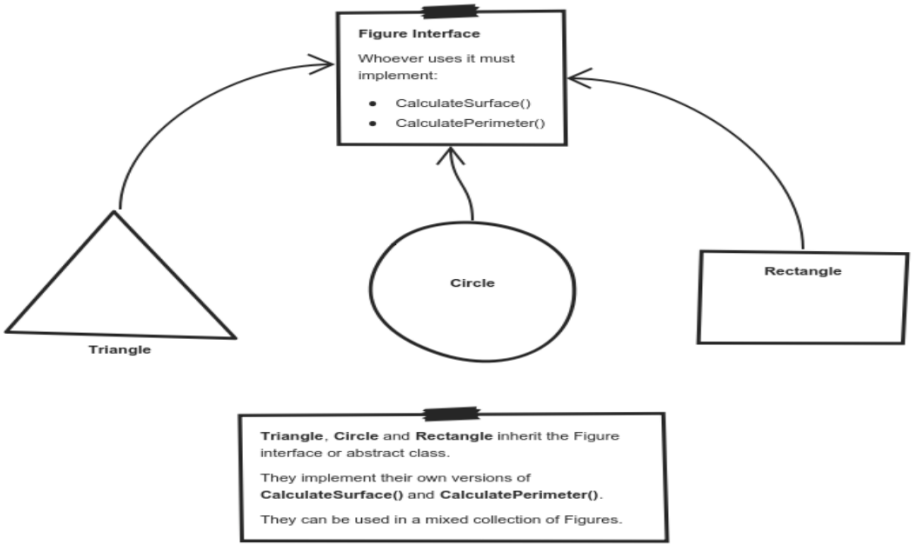
II.4 Polymorphism

We are down to the most complex word!Polymorphism means “many shapes” in Greek. So we already know the power of inheritance and happily use it. But there comes this problem.

Say we have a parent class and a few child classes which inherit from it. Sometimes we want to use a collection, for example a list, which contains a mix of all these classes. Or we have a method implemented for the parent class, but we’d like to use it for the children, too.

This can be solved by using polymorphism. Simply put, polymorphism gives a way to use a class exactly like its parent so there’s no confusion with mixing types. But each child class keeps its own methods as they are.

This typically **happens by defining a (parent) interface to be reused.** It outlines a bunch of common methods. Then, each child class implements its own version of these methods.Any time a collection (such as a list) or a method expects an instance of the parent (where common methods are outlined), the language takes care of evaluating the right implementation of the common method regardless of which child is passed.Take a look at a sketch of geometric figures implementation. They reuse a common interface for calculating surface area and perimeter:



Having these three figures inheriting the parent Figure Interface lets you create a list of mixed triangles, circles, and rectangles.And treat them like the same type of object. Then, if this list attempts to calculate the surface for an element, the correct method is found and executed. If the element is a triangle, triangle's CalculateSurface() is called. If it's a circle, then cirlce's CalculateSurface() is called. And so on.

If you have a function which operates with a figure by using its parameter, you don't have to define it three times, once for a triangle, a circle, and a rectangle.

You can define it once and accept a Figure as an argument. Whether you pass a triangle, circle or a rectangle, as long as they implement CalculateParamter(), their type doesn't matter.

So , polymorphism refers to the ability to perform a certain action in different ways. In Java, polymorphism can take two forms: method overloading and method overriding. Method overloading happens when various methods with the same name are present in a class. When they are called they are differentiated by the number, order, and types of their parameters. Method overriding occurs when the child class overrides a method of its parent.

II.4.1 Polymorphism in Java:

1. The same method name is used several times.
2. Different methods of the same name can be called from the object.
3. All Java objects can be considered polymorphic (at the minimum, they are of their own type and instances of the Object class).
4. Example of static polymorphism in Java is method overloading.
5. Example of dynamic polymorphism in Java is method overriding.

Check more https://www.javatpoint.com/runtime-polymorphism-in-java

Chapter III

Goals

- Review main OOP Concepts
- Understand how Abstraction works
- Understand how Encapsulation works
- Understand how Polymorphism works
- Understand how Inheritance works
- Practice using OOP principles

Chapter IV

Exercise 00 :Inheritance

Using inheritance, one class can acquire the properties of others. Consider the following Animal class:

```
class Animal{
    void walk(){
     ...
    }
}
```

This class has only one method, walk. Next, we want to create a Bird class that also has a fly method. We do this using extends keyword:

```
class Bird extends Animal {
    ...
}
```

Finally, we can create a Bird object that can both fly and walk and do other things.

```
public class Main{
   public static void main(String[] args){

      Bird sparrow = new Bird();
      ...
     sparrow.fly();
      ...
   }
```

The above code will print:

```
I am walking
I am flying
I am singing
```

Chapter V

Exercise 01 : Inheritance

Write the following code in your editor below:

- A class named Arithmetic with a method named add that takes integers as parameters and returns an integer denoting their sum.
- A class named Adder that inherits from a superclass named Arithmetic.

Input Format
You are responsible for reading any input from stdin
Output Format
Your method must return the sum of its parameters.

V.0.1 Java Stdin and Stdout

http://javaispapa.blogspot.com/2017/07/java-stdin-and-stdout.html

```
public class Main{
   public static void main(String[] args){

    // Create a new Adder object
        Adder a = new Adder();
    // Print the result of 2 calls to Adder's add(int,int) method as 3 space-separated integers:
        System.out.println(a.add(10,32));
        System.out.println(a.add(5,15));
   }
```

The above code will print:

```
42
20
```

Chapter VI

Exercise 02 :OOP

When a subclass inherits from a superclass, it also inherits its methods; however, it can also override the superclass methods (as well as declare and implement new ones). Consider the following Sports class:

```
class Sports{
    String getName(){
        return ("Generic Sports");
    }
    void getNumberOfTeamMembers(){
        System.out.println("Each team has n players in" + getName());
    }
```

Next, we create a Soccer class that inherits from the Sports class. We can override the getName method and return a different, subclass-specific string:

```
class Soccer extends Sports{
    @Override
    String getName(){
        return ("Soccer Class");
    }
```

VI.0.1

Note: When overriding a method, you should precede it with the @Override annotation. The parameter(s) and return type of an overridden method must be exactly the same as those of the method inherited from the supertype.

Task Complete the code in your editor by writing an overridden getNumberOfTeam-Members method that prints the same statement as the superclass' getNumberOfTeam-Members method, except that it replaces with (the number of players on a Soccer team). Output Format When executed, your completed code should print the following:

```
Generic Sports
Each team has n players in Generic Sports
```

```
Soccer Class
Each team has 11 players in Soccer Class
```

Chapter VII

Exercise 03 :OOP

We are going to go back to the car analogy.

- Create a base class called Car
- It should have a few fields that would be appropriate for a generice car class(boolean engine, int cylinders, int wheels,String model)
- Constructor should initialize cylinders (number of) and name, and set wheels to 4 and engine to true. Cylinders and names would be passed parameters.
- Create appropriate getters
- Create some methods like startEngine, accelerate, and brake show a message for each in the base class
- Now create 3 sub classes (Mitsubishi, Volvo, Mercedes)
- Override the appropriate methods to demonstrate polymorphism in use

Example :

```
public class Main{
   public static void main(String[] args){
    Car car = new Car(8, /"Base car");
        System.out.println(car.startEngine());
        System.out.println(car.accelerate());
        System.out.println(car.brake());
        }
        }
```

```
Car -> startEngine()
Car -> accelerate()
Car -> brake()
```

```
public class Main{
   public static void main(String[] args){
       // type your favorite model, mine is Outlander VRX 4WD
    Mitsubishi mitsubishi = new Mitsubishi(6, /"Outlander VRX 4WD");
        System.out.println(mitsubishi.startEngine());
        System.out.println(mitsubishi.accelerate());
        System.out.println(mitsubishi.brake());
        }
        }
```

```
Mitsubishi -> startEngine()
Mitsubishi -> accelerate()
Mitsubishi -> brake()
```