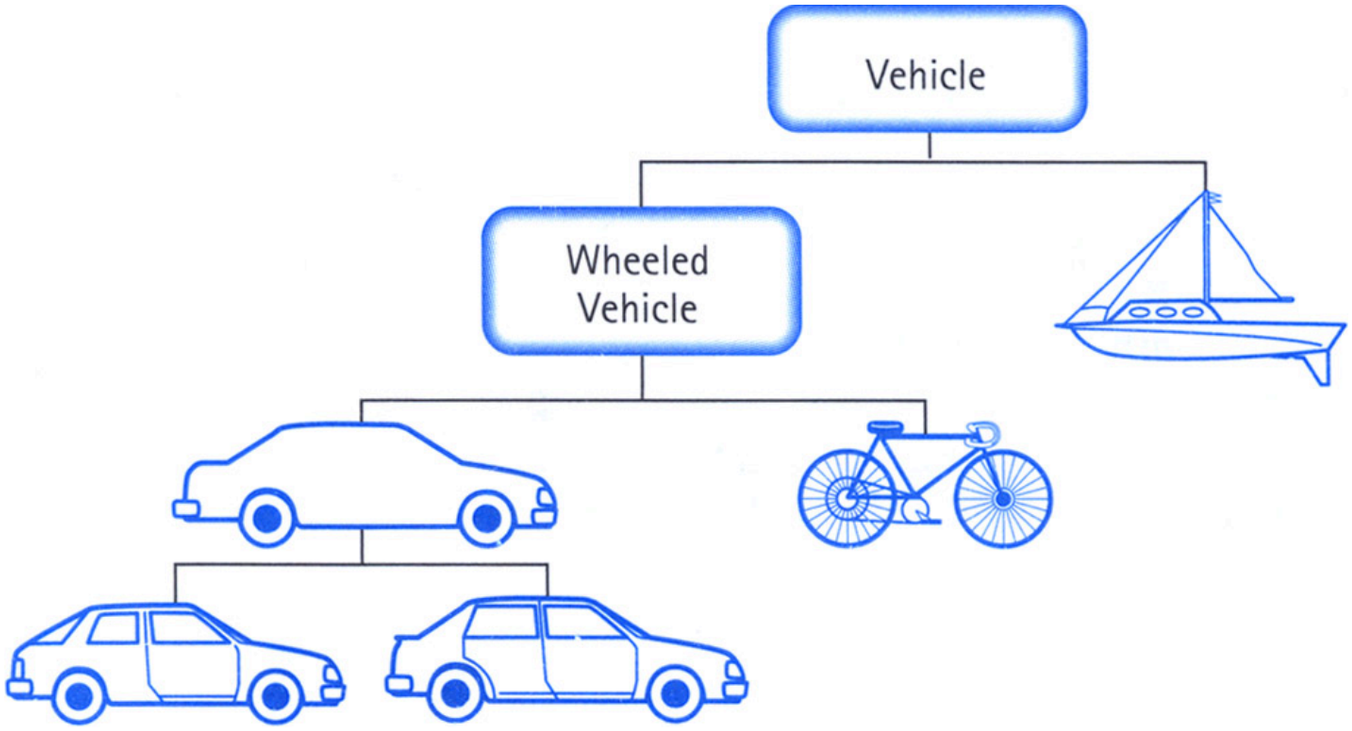
Object Oriented Design



Decomposing programs

In many languages (e.g. C), programs are decomposed into functions, that operate on common data structures.

This is called ***functional decomposition***

Functional decomposition

Cons:

Modern systems perform **more than one function**

Systems evolve, their functions change

Object Oriented Decomposition

A system is decomposed according to the **objects** a system is supposed to manipulate.

OO Concepts

There are 3 core concept at the heart of OO:

1. Encapsulation
2. Inheritance
3. Polymorphism

Encapsulation

Group together data (variables) and methods (functions) in one unit.

Also, all variables should be **hidden** (private) and only accessible by the methods in the class.

Classes

A class is a template for creating objects.

Example: a car

it has two attributes: brand name and fuel level

and two methods: drive and refuel

```java
public class Car {

    private String brandName;
    private double fuelLevel;

    public Car(String brandName) {
        t
        f
    }

    publi
        f
    }

    publi
        fuelLevel = 10;
    }
}
```

These an attribute of the class.

In Java, attributes they are known as **fields.**

The **private** keyword specifies that the attribute is only accessible by the method of that class.

```java
public class Car {

    private String brandName;
    private double fuelLevel;

    public Car(String brandName) {
        this.brandName = brandName;
        fuelLevel = 10;
    }

    public vo
        fuelL
    }

    public vo
        fuelL
    }
}
```

This is the **constructor**.

It is used for creating objects, with the *new* keyword

The **this** keyword disambiguates between the field and parameter.

```
public class Car {

    private String brandName;
    private doubl

    public Car(St
        this.bran
        fuelLevel

    }

    public void drive() {
        fuelLevel = fuelLevel - 1;
    }

    public void refuel() {
        fuelLevel = 10;
    }
}
```

These are **methods**.

Methods are operations that this object can perform

Information hiding

The **private** keyword is used to keep all data hidden

But what if I want to access, or to change, the value outside of a class?

We define special methods, **getters** and **setters**

Only define getters and setters if you need them!

```
public double getFuelLevel() {
    return fuelLevel;
}
```

```
public void setBrandName(String brandName) {
    this.brandName = brandName;
}
```

A short question

Creating objects

Objects are created with the **new** keyword

```
Car c = new Car("Ford")
```

This invokes the constructor with the right parameters.

Inheritance

Also known as **subclassing** or **subtyping**

Classes can inherit fields and methods from other classes with the **extends** keyword.

We want to model a Sedan, that has all the fields and methods of a person.

Defines a "is-a" relationship between classes.

```
public class Sedan extends Car {
    private int
    public Sedan
        super(name);
    }
}
```

The class declaration now contains the **extends** declaration

The constructor now contains the **super** keyword. **This passes the parameters to Car's constructor.**

```
public cla

    private int noOfDoors = 4;

    public Sedan(String name) {
        super(name);
    }
}
```

```
public Car(String brandName) {
    super(brandName);
}
```

Inheritance

`Sedan` now inherits `Car`'s attributes and method:

```
Sedan s = new Sedan("Ford");
s.drive();
```

Polymorphism

Polymorphism means taking different forms

In Java, this refers to the fact that a subclass can always be used instead of a parent class.

e.g. You can use a **`Sedan`** object, even if a **`Car`** is required:

```
Car c = new Sedan("Ford");
```

Class hierarchies

We want to model a boat. It has a brand name, a fuel level, but it cannot drive.

We can create an **abstract** class, `Vehicle`, from which we can extend for `Car` and `Boat`

```java
public class Vehicle {

    private String brandName;
    protected double fuelLevel;

    public void refuel() {
        fuelLevel = 10;
    }

    public double getFuelLevel() {
        return fuelLevel;
    }

    public void setBrandNa
        this.brandName = b
    }
}
```

{

The **protected** keyword allows subclasses to access this field

We **extracted** all the common functionality between Car and Boat (the name and the fuel) into it's own class

```java
public class Car extends Vehicle {

    public Car(String brandName) {
        super(brandName);
    }

    public void drive() {
        fuelLevel = fuelLevel - 1;
        // some other code that "drives" the car
    }

}
```

We access the protected field in `Vehicle`

```java
public class Boat extends Vehicle {

    public Boat(String name) {
        super(name);
    }

    public void sail() {
        fuelLevel = fuelLev
        // some code relati
    }
}
```

The class **`Boat`** now has to deal only with Boat specific stuff

What's the advantage?

It allows us to write code that is more generic

```
public void refuel(Vehicle v) {
    v.refuel();
}
```

Dynamic polymorphism

This will work with any vehicle.

It keeps the code clean, and easy to maintain.

Method overloading

In Java, multiple methods can have the same name, as long they have different parameters (type and/or numbers)

```
public void refuel() {
    fuelLevel = 10;
}

public void refuel(int x) {
    fuelLevel = x;
}
```

Static polymorphism