Inf1B

Classes vs. Objects

Volker Seeker

School of Informatics

January 24, 2020

Why OO?

Software engineering as managing change

Changing code is hard and expensive, but because the world changes, essential.

Software engineering as managing change

How can we make changing code easy and cheap?

- minimise the amount of code that must change
- make it easy to work out which code must change

→ have the code that must change live together

How can we make change easier and cheaper?

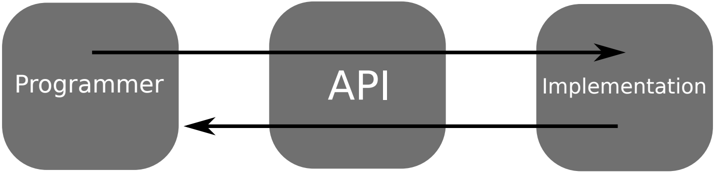
Key idea: Information Hiding

Hide certain information inside well-defined pieces of code, so that users of that piece of code don't depend on it, and don't need to change if it changes.

→ e.g. Modularity and Abstraction via Functions

Application Programming Interface

The interface between the user of the code and the implementation itself is called an Application Programming Interface (API).



Intuition

Client

client needs to know how to use API

API

- adjust volume
- switch channel
- switch to standby

Implementation

- cathode ray tube
- 20” screen, 22 kg
- Sony Trinitron KV20M10

implementation needs to know what API to implement

Implementation and client need to agree on API ahead of time.

Intuition

Client

client needs to know how to use API

API

- adjust volume
- switch channel
- switch to standby

Implementation

- HD LED display
- 37” screen, 10 kg
- Samsung UE37C5800

implementation needs to know what API to implement

Can substitute better implementation without changing the client.

Data representation

Recall: a data type is a set of values and operations on those values. May be

- primitive, built into the language with operations defined in the compiler/runtime, e.g. `int`, `double`, `boolean`
- user-defined, with operations defined in the programming language itself, e.g. `PrinterQueue`, `HotelRoom`, . . .

Data representation

Recall: a data type is a set of values and operations on those values. May be

- primitive, built into the language with operations defined in the compiler/runtime, e.g. `int`, `double`, `boolean`
- user-defined, with operations defined in the programming language itself, e.g. `PrinterQueue`, `HotelRoom`, . . .
- Intermediate case where user-defined types are provided with the standard libraries in Java, e.g. `String`.

Hiding data representation

You shouldn't need to know how a data type is implemented in order to use it. It should suffice to read the documentation: what operations are there, what do they do?

→ Then you can write code that won't need to change if the implementation changes.

This concept is known as **Encapsulation**

The general idea is not specific to OO, but Java does it differently from Haskell.

Towards object oriented programming...

So far in this course, we've been doing
Procedural programming [verb oriented]

- tell the computer to do this, then
- tell the computer to do that.

You know:

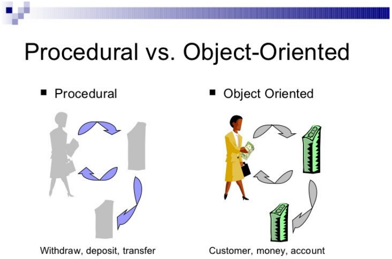
- how to program with primitive data types e.g. `int`, `boolean`;
- how to control program flow to do things with them, e.g. using `if`, `for`;
- how to group similar data into arrays.

Philosophy of object orientation

Problem: what your software must do changes a lot. Structuring it based on that is therefore expensive.

The **domain** in which it works changes much less.

→ structuring your software around the **things** in the domain makes it easier to understand and maintain.



Procedural vs. Object-Oriented

- Procedural
- Object Oriented

Withdraw, deposit, transfer

Customer, money, account

https://www.alphansotech.com/wp-content/uploads/2015/11/object-oriented-concept-13-728-1.jpg

Philosophy of object orientation

Object Oriented programming (OOP)

[noun oriented]

- Things in the world know things: instance variables.
- Things in the world do things: methods.

In other words, objects have state and behaviour.

State and Behaviour

State

- running (yes/no)
- speed (10mph)
- petrol (87%)

Behaviour

- start Engine
- stop Engine
- accelerate
- break
- refill petrol

State and Behaviour

State

- running (yes/no)
- speed (10mph)
- petrol (87%)

Behaviour

- start Engine
- stop Engine
- accelerate
- break
- refill petrol

A program runs by objects sending messages (initiating behaviour) to one another, and reacting to receiving messages (e.g. changing state, sending more messages).

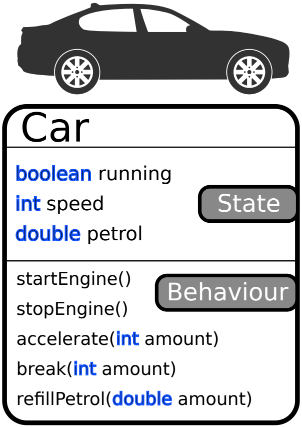
Classes and Objects

How does this work in Java?

Classes to organise code

Java is a **class-based object-oriented** language.

All code is organised in classes which serve as user defined data types.



Car

boolean running
int speed
double petrol

State

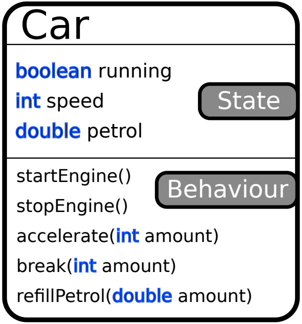
startEngine()
stopEngine()
accelerate(int amount)
break(int amount)
refillPetrol(double amount)

Behaviour

Classes to organise code

Java is a **class-based object-oriented** language.

All code is organised in classes which serve as user defined data types.



Car

boolean running

int speed

double petrol

State

startEngine()

stopEngine()

accelerate(int amount)

break(int amount)

refillPetrol(double amount)

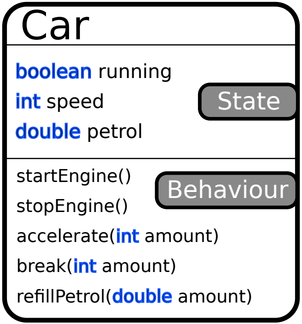
Behaviour

All the classes you wrote so far only defined behaviour.

Creating a class instance

Now only one important thing is missing.

A **Constructor**.



Car

boolean running
int speed
double petrol

State

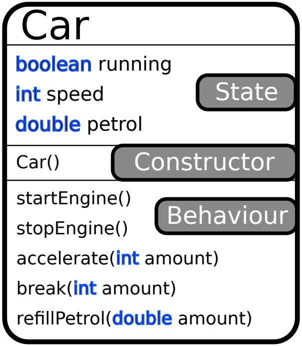
startEngine()
stopEngine()
accelerate(int amount)
break(int amount)
refillPetrol(double amount)

Behaviour

Creating a class instance

Now only one important thing is missing.

A **Constructor**.



Car

boolean running

int speed

double petrol

State

Car()

Constructor

startEngine()

stopEngine()

accelerate(int amount)

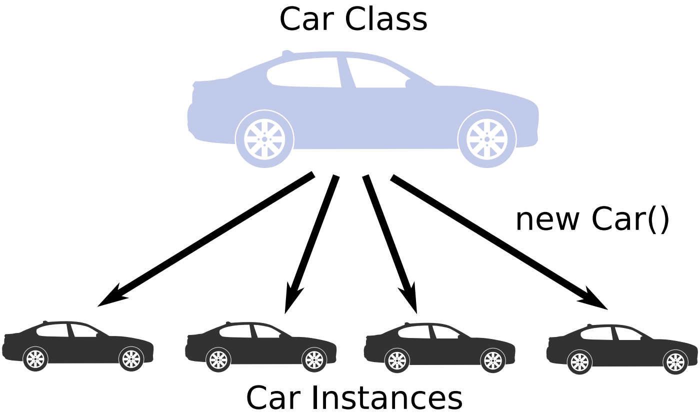
break(int amount)

refillPetrol(double amount)

Behaviour

A Constructor is used to create an instance of a class which can then be used in your program.

Classes as blueprints



Car Class

new Car()

Car Instances

- Constructor is a special method with the same name as the class
- Allocates memory for the class instance and initialises its state

Instances are Objects

In Java, instances of classes are what you consider to be **Objects**.

Car Example

Using a Car class and its API

```
Car mycar = new Car();
mycar.startEngine();
mycar.accelerate(30);
mycar.break(30);
mycar.stopEngine();
mycar.refillPetrol(0.5);
```

Car Example

Using a Car class and its API

```
Car mycar = new Car();
mycar.startEngine();
mycar.accelerate(30);
mycar.break(30);
mycar.stopEngine();
mycar.refillPetrol(0.5);
```

Note that we have two independent ideas here:

- Conceptual objects (class instances) such as `mycar` are directly present in the program;
- They have static (compile-time) types (Car class) that define their behaviour.

Objects ...

- have a static (compile-time) type defined inside a class
- are instances of classes created at runtime
- are created using a constructor and the new keyword

Objects ...

- have a static (compile-time) type defined inside a class
- are instances of classes created at runtime
- are created using a constructor and the new keyword
- are reference types

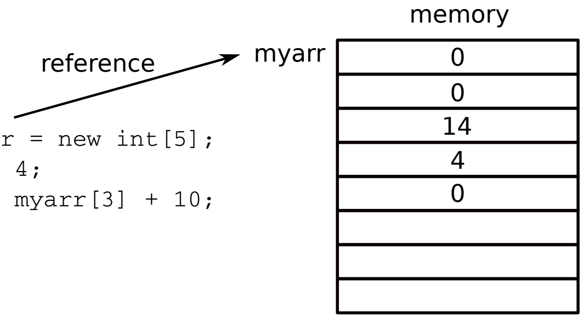
Objects are Reference Types

What happens in memory?

Arrays in Memory

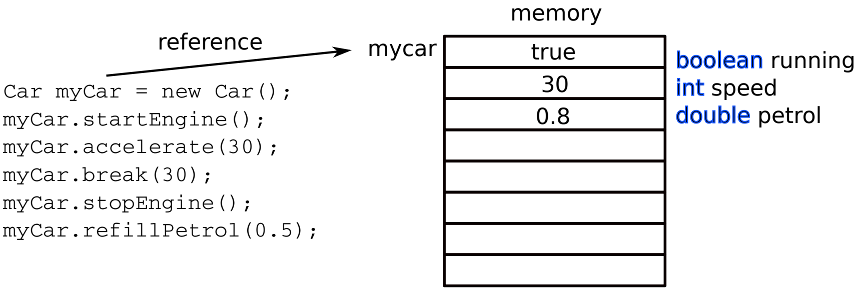
Recall what happens with arrays:

```
int[] myarr = new int[5];
myarr[3] = 4;
myarr[2] = myarr[3] + 10;
```



Class instances in memory

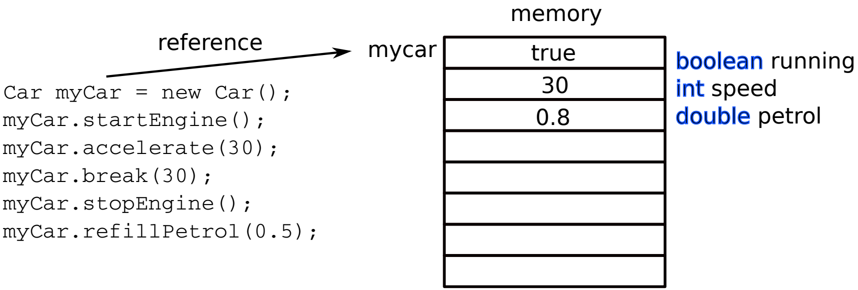
What happens to our Car?



```
Car myCar = new Car();
myCar.startEngine();
myCar.accelerate(30);
myCar.break(30);
myCar.stopEngine();
myCar.refillPetrol(0.5);
```

Class instances in memory

What happens to our Car?



```
Car myCar = new Car();
myCar.startEngine();
myCar.accelerate(30);
myCar.break(30);
myCar.stopEngine();
myCar.refillPetrol(0.5);
```

reference

mycar

memory

memory	
true	boolean running
30	int speed
0.8	double petrol

- creating a class instance reserves memory for its state (plus some interal extras)
- the constructor is executed to initialise this memory (hence new and ctor in combination)
- the local variable `mycar` holds a reference to the actual object representation in memory (same as for arrays)

Closing the Loop on Arrays

The Java language specification states:

An object is a class instance or an array.

In Java, arrays are treated like class instances, e.g.

- created using `new`
- referenced in memory
- underlying class definintion (hidden in the language implementation).

However, they differ in some ways, e.g.

- in the way their state is accessed: `myarr[3] = 5;`
- except for `length`: `for(int i =0; i < myarr.length; i++)`
- and have no behaviour methods.

What happens for uninitialised objects?

```
Car myCar;
myCar.startEngine();
```

What happens for uninitialised objects?

```
Car myCar;
myCar.startEngine();
```

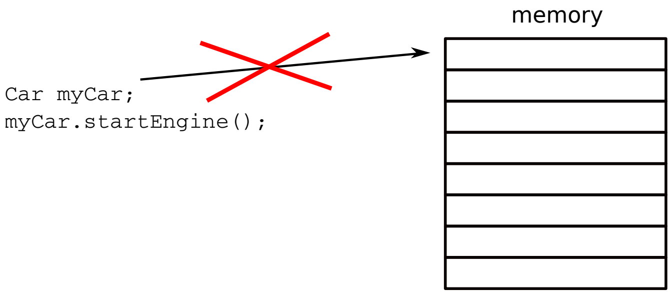
This will throw a java.lang.NullPointerException at runtime since no memory was allocated for *myCar* using the `new` operator.

What happens for uninitialised objects?

```
Car myCar;
myCar.startEngine();
```

This will throw a java.lang.NullPointerException at runtime since no memory was allocated for *myCar* using the `new` operator.

No reference, no memory allocated:



```
Car myCar;
myCar.startEngine();
```

memory

Referencing nothing

Where do references point when there is no corresponding object allocated for them?

Referencing nothing

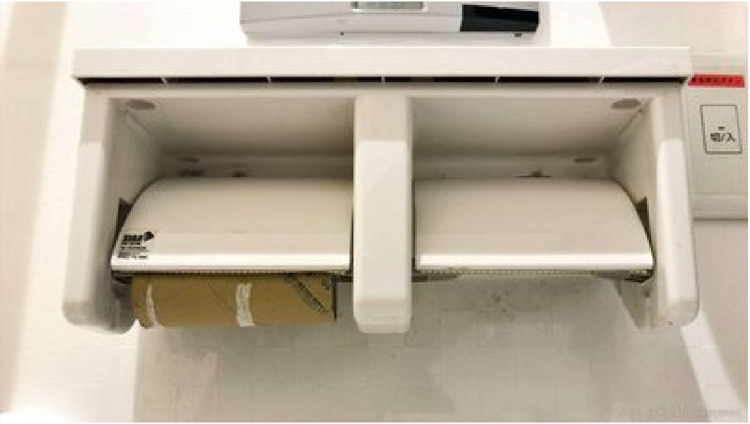
Where do references point when there is no corresponding object allocated for them?

```
Car myCar = null;
```

The null literal indicates an object reference pointing at nothing.

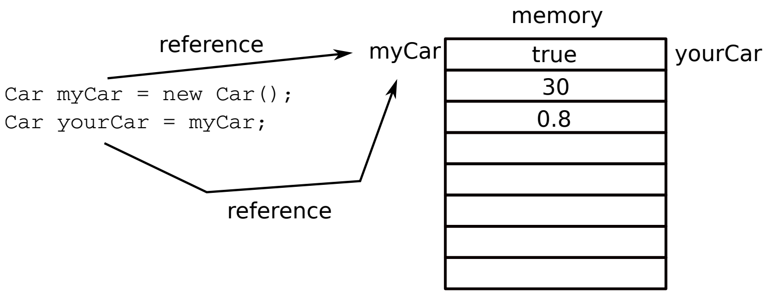
Using the `myCar` variable to call a method on it or change its state will now result in a **java.lang.NullPointerException**.

Null - Know the difference!



Class instances in memory

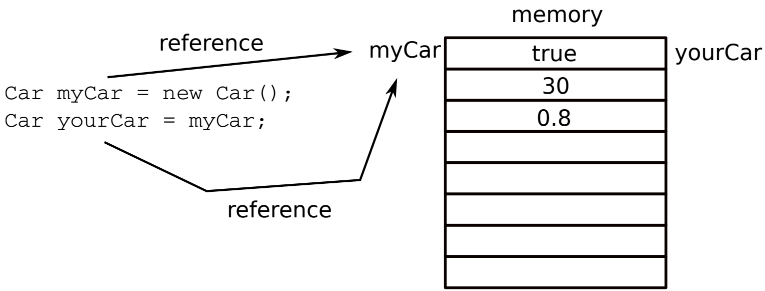
Copying an object instance:



```
Car myCar = new Car();
Car yourCar = myCar;
```

Class instances in memory

Copying an object instance:



```
Car myCar = new Car();
Car yourCar = myCar;
```

reference

reference

myCar

yourCar

memory

true

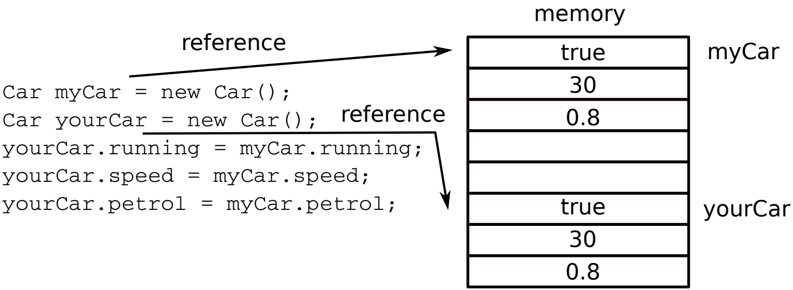
30

0.8

Assigning the reference of an object instance to a local variable of the same type does **not** copy the object's memory, only its reference!

Class instances in memory

Copying an object instance:



```
Car myCar = new Car();
Car yourCar = new Car();
yourCar.running = myCar.running;
yourCar.speed = myCar.speed;
yourCar.petrol = myCar.petrol;
```

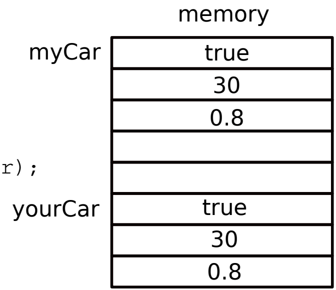
memory
true
30
0.8
true
30
0.8

To copy an instance, a new one of the same type needs to be created and its entire state copied over.

Class instances in memory

Comparing class instances:

```
Car myCar = new Car();
Car yourCar = new Car();
System.out
 .println(myCar == yourCar);
```

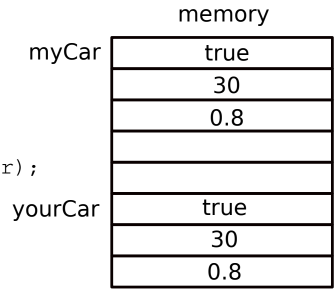


What does this print?

Class instances in memory

Comparing class instances:

```
Car myCar = new Car();
Car yourCar = new Car();
System.out
 .println(myCar == yourCar);
```



memory

myCar

true

30

0.8

yourCar

true

30

0.8

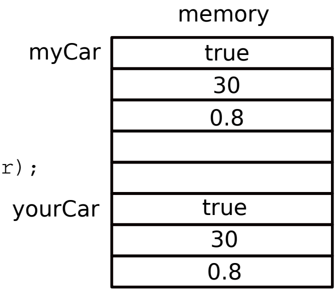
What does this print?

False

Class instances in memory

Comparing class instances:

```
Car myCar = new Car();
Car yourCar = new Car();
System.out
 .println(myCar == yourCar);
```



memory

myCar

true

30

0.8

yourCar

true

30

0.8

What does this print?

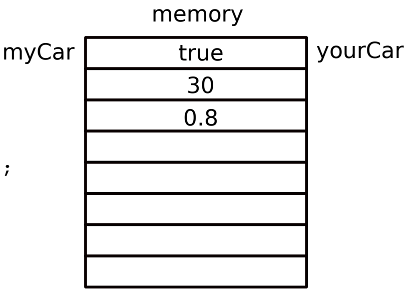
False

== compares object references not object states

Class instances in memory

Comparing class instances:

```
Car myCar = new Car();
Car yourCar = myCar;
System.out
 .println(myCar == yourCar);
```



memory

myCar

yourCar

true

30

0.8

What does this print?

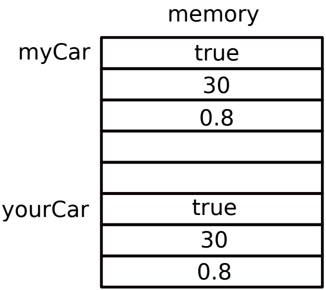
True

== compares object references not object states

Class instances in memory

Comparing class instances:

```
Car myCar = new Car();
Car yourCar = new Car();
System.out
 .println(myCar.speed ==
           yourCar.speed);
```



memory

myCar

true

30

0.8

yourCar

true

30

0.8

What does this print?

True

== compares object references not object states

in contrast to primitive types

Class instances in memory

Comparing class instances:

Conveniently, most classes coming with the Java library such as `String` or `Integer` implement the comparison method **equals**.

```
Integer sizeA = new Integer(700);
Integer sizeB = new Integer(700);

// prints true
System.out.println(sizeA.equals(sizeB));
```

By convention, the equals method is implemented in a way that compares the states of two objects. (Later I will show you how you can do that for your own types.)

Let's practice that

https://www.theodysseyonline.com/your-brain-is-muscle-exercise-it

Class vs Instance Methods

Using methods

Using a method associated with an instance of a class

```
Car myCar = new Car();
myCar.startEngine();
myCar.accelerate(20);
```

The method is called by using the ’.’ operator on the variable name of the class instance.

Using methods

Using a method associated with an instance of a class

```
Car myCar = new Car();
myCar.startEngine();
myCar.accelerate(20);
```

The method is called by using the '.' operator on the variable name of the class instance.

But what about this?

```
double rnd = Math.random()* 10;
```

Here, the method is called by using the '.' operator on the class name itself.

Class Methods vs. Instance Methods

Instance Methods:

- Associated with an object.
- Identifying an instance method requires an object name: `myCar.startEngine()`

Class Methods:

- Associated with a class.
- Identifying a method in a separate class requires name of the class: `Math.random()`.

Class Methods vs. Instance Methods

Consider class methods to be globally available, should you be able to import the corresponding type.

They are also called `static` methods indicated by the function modifier you need to use when implementing them.

There is not just static behaviour, there is also static state or static constants.

Global Constants

Similar to globally available class methods, global constants can be declared and initialised using the `static` and `final` keywords.

```java
public class MathHelper {

    public static final double PI = 3.141592653589793;

    // ... some helpful math functions
}

public class Main {

    public static double circleArea(double radius) {
        return MathHelper.PI * radius * radius;
    }
}
```

Summary

Summary: Why use object orientation?

OO has taken the world by storm. Why?

Summary: Why use object orientation?

OO has taken the world by storm. Why?
It is well suited to support good *software engineering* practices.

Quick reminder: this is not a SE course, however, it lays the foundation for it.

Summary: Why use object orientation?

OO has taken the world by storm. Why?
It is well suited to support good *software engineering* practices.

Quick reminder: this is not a SE course, however, it lays the foundation for it.

- use objects to model real-world entities
- use classes to model domain concepts.
- These change more slowly than specific functional requirements,
- so what OO does is to put things together that change together as requirements evolve.

Change is the thing that makes software engineering hard and interesting; OO helps manage it.

Summary: in Java

- A variable can have
  - a primitive type e.g., `boolean`, `int`, `double`; or
  - a reference type: any class, e.g. `String`, `Car`, `Color` and any array type.
- Instances of reference types are created using `new`.
- Variables of reference types contain references to their representation in memory.
  - Two references can refer to the same memory location.
  - Copying the reference does not copy the state of the object
  - `==` compares references, `.equals` compares state.
- Lastly, object behaviour can be expressed by using class and instance methods.

Reading

Objects First

Chapter 1
Note that this book uses *BlueJ* which is a specialised IDE for teaching Object Oriented programming. Feel free to use it as well if you want to go over the exercises.