Inf1B

Classes and Objects

Fiona McNeill
adapting earlier versions by Perdita Stevens, Ewan Klein, Volker Seeker, et al.

School of Informatics

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

$\rightarrow$ have the code that must change live together

How can we make change easier and cheaper?

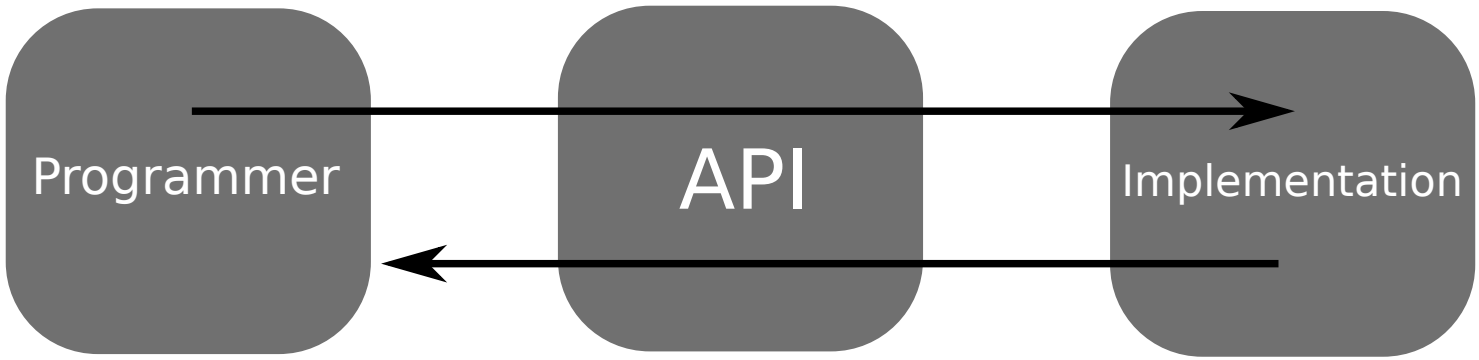
Key idea: Information Hiding

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

API

- adjust volume
- switch channel
- switch to standby

Implementation

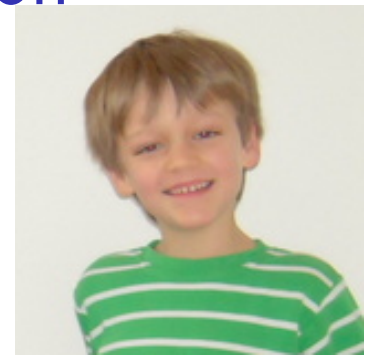
- cathode ray tube
- 20" screen, 22 kg
- Sony Trinitron KV20M10

client needs to know how to use API

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 some really important types are not primitive, but 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.

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?

Terminology:

- abstraction: you don't need to know about implementation details
- encapsulation: you can't depend on implementation details

Different languages have different mechanisms.

Towards object oriented programming...

So far in this course, we've been doing

Procedural programming

- 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.

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

Philosophy of object orientation

- 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).

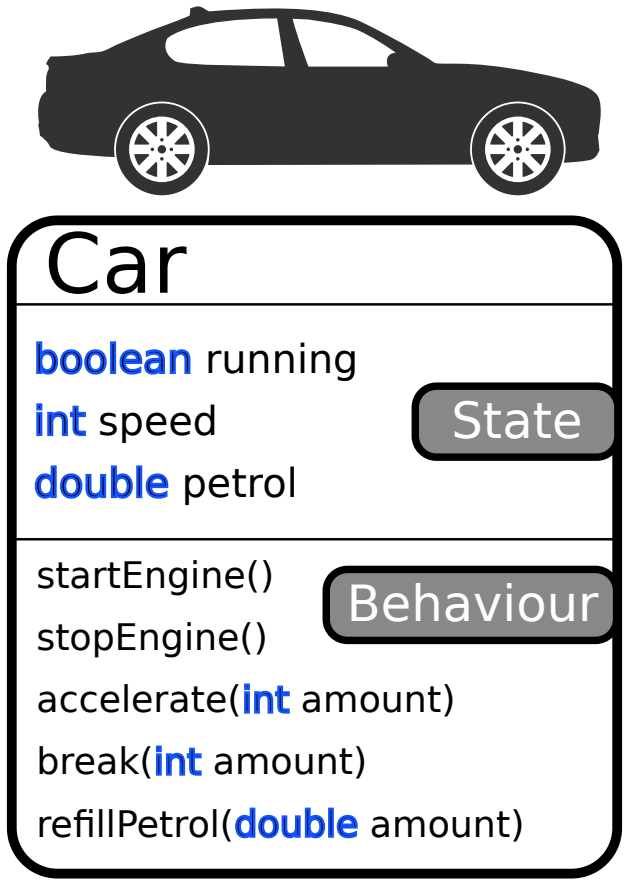
Classes and Objects

How does this work in Java?

Classes to organise code

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

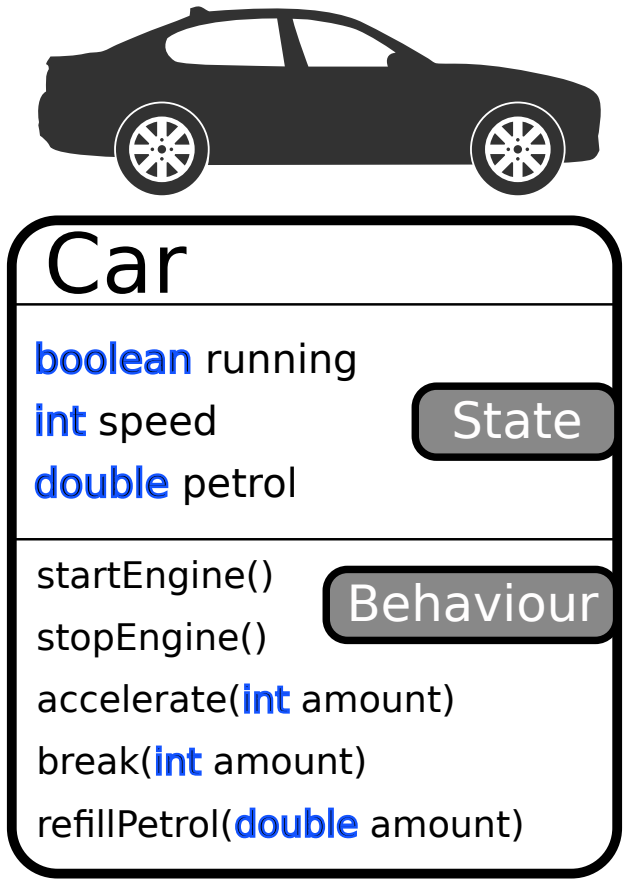
All code is organised in classes which serve as user defined data types.



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.

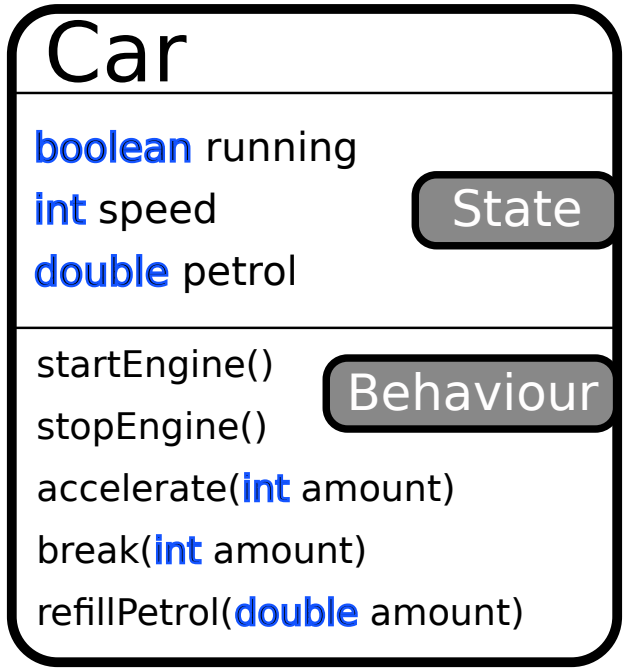


All the classes you wrote so far only defined behaviour.

Creating a class instance

Now only one important thing is missing.

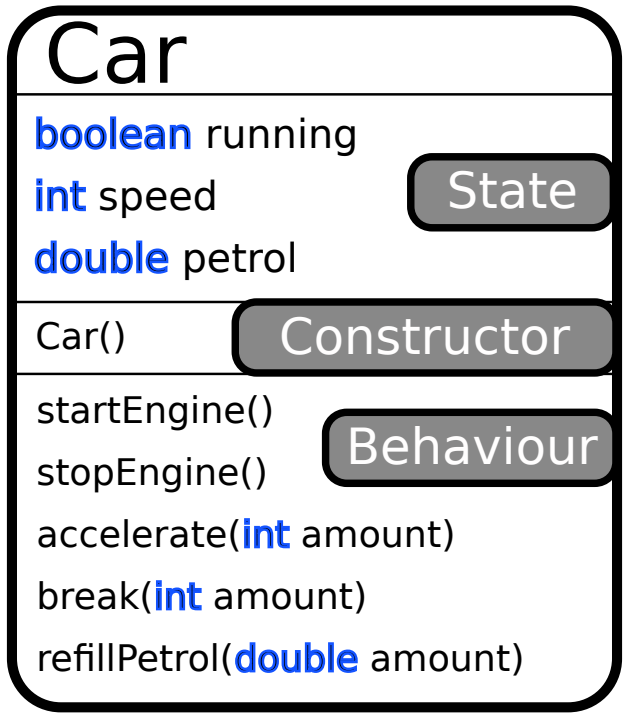
A **Constructor**.



Creating a class instance

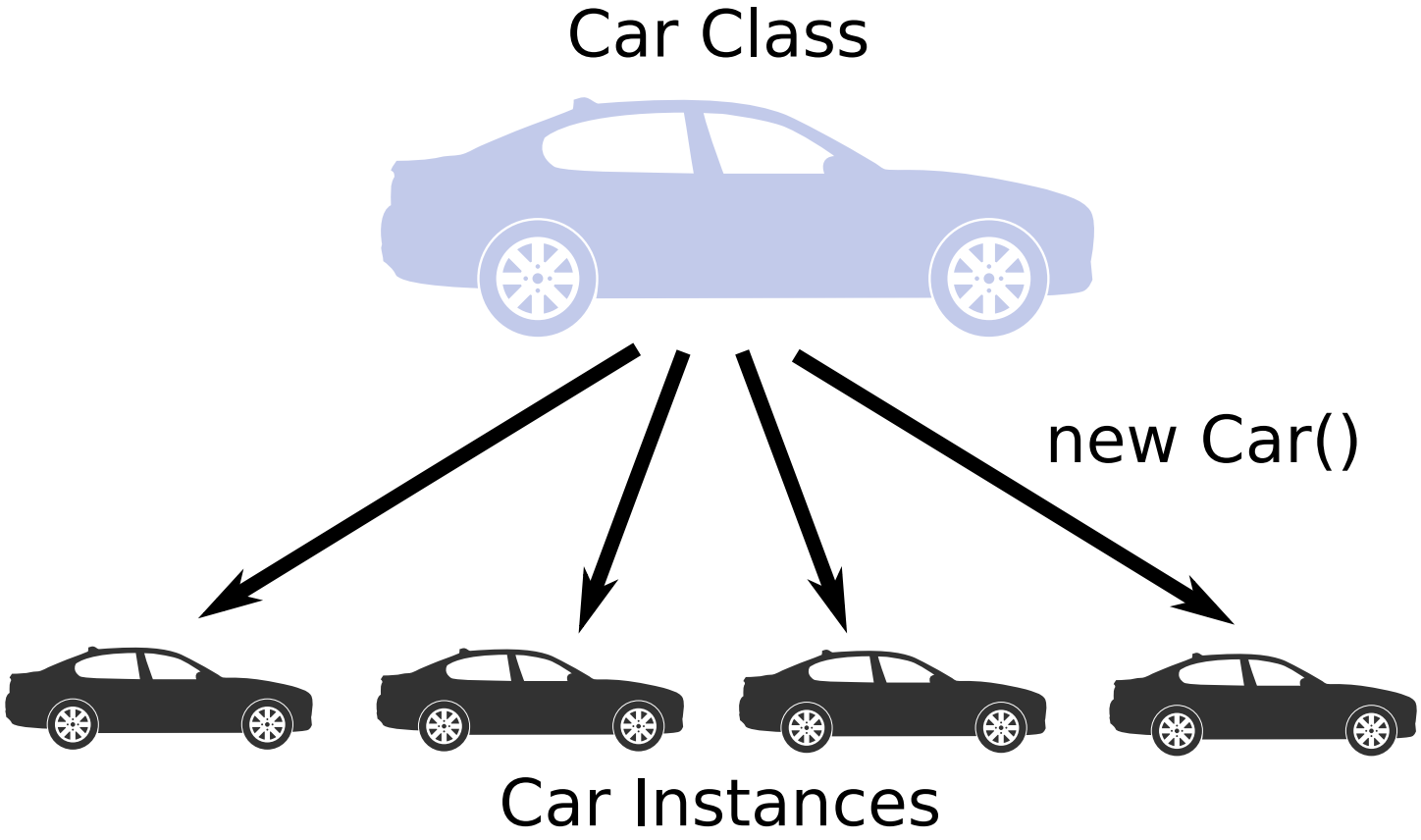
Now only one important thing is missing.

A **Constructor**.



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

Classes as blueprints



- 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 **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 ...

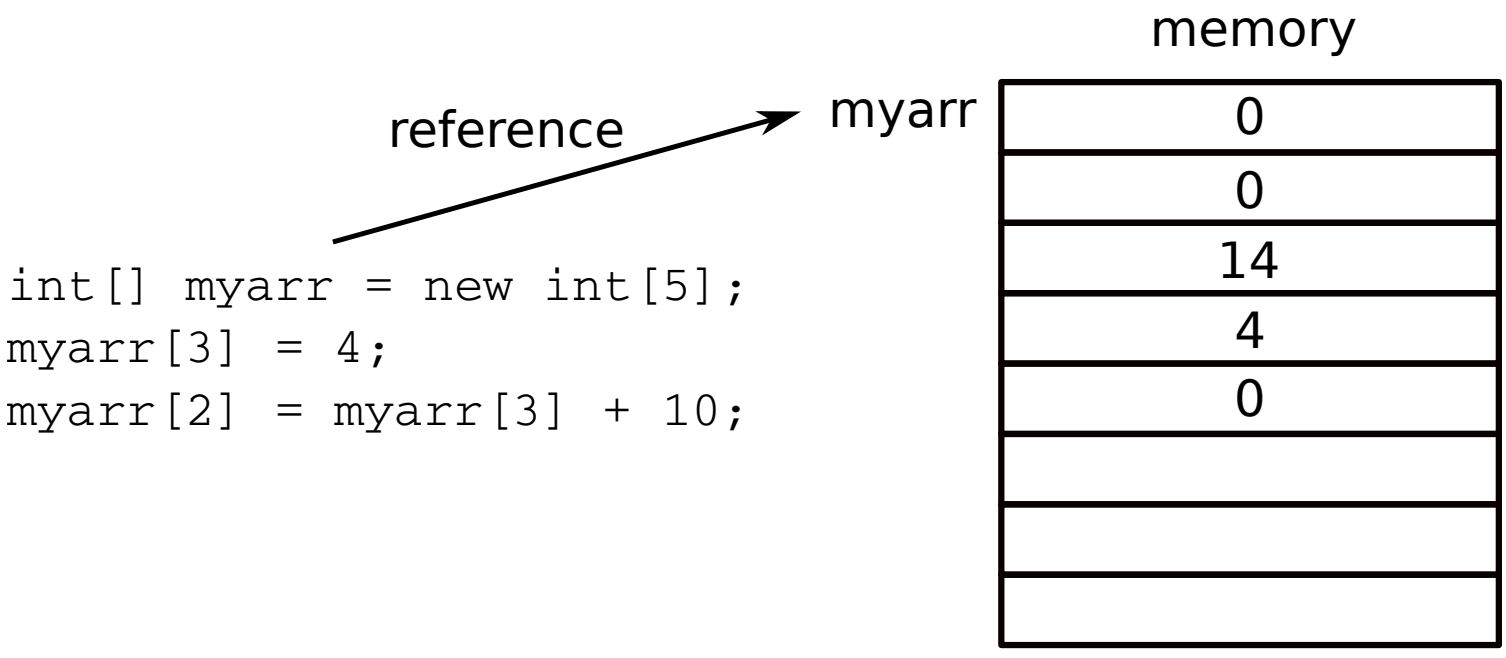
- have a static (compile-time) type defined inside a class
- are instances of classes created at runtime
- are created using a constructor and the new keyword
- are reference types

Objects are Reference Types

What happens in memory?

Arrays in Memory

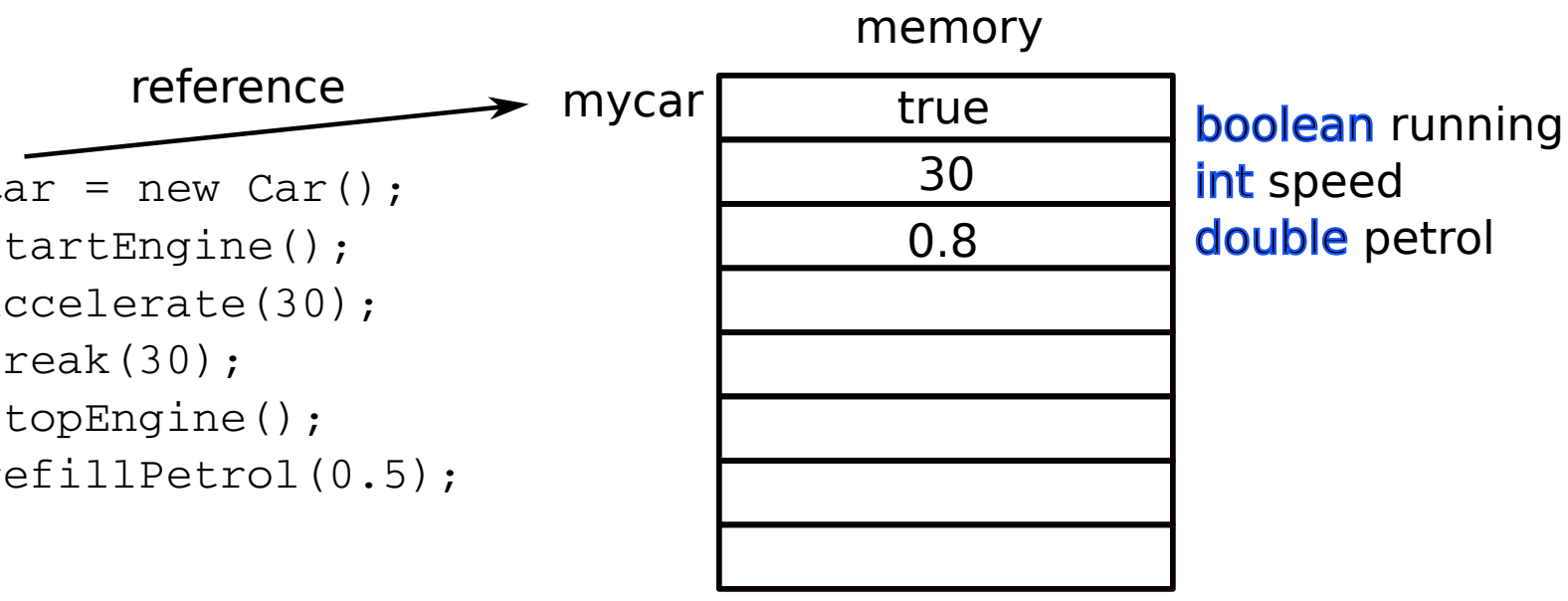
Recall what happens with arrays:



Class instances in memory

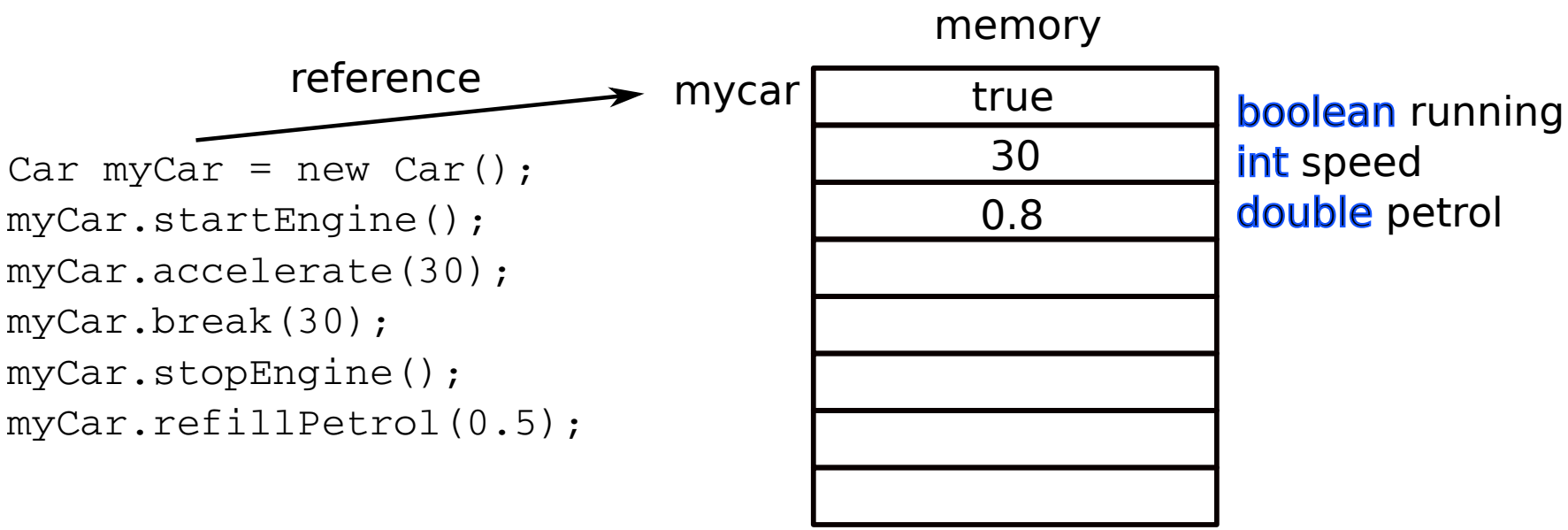
What happens to our Car?

```
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?



- creating a class instance reserves memory for its state (plus some internal extras)
- the constructor is executed to initialise this memory (hence new and constructor 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 definition (hidden in the language implementation).

However, they differ a lot, e.g.

- special way to access state: myarr[3] = 5;
- special way to get `length`:
  for (int i =0; i < myarr.length; i++)
- no methods.

What happens for uninitialised objects?

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

What happens for uninitialised objects?

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

This will not compile: you'll get an error

`error: variable myCar might not have been initialized`

No memory has been allocated for it (using new); no object has been created.

What happens for uninitialised objects?

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

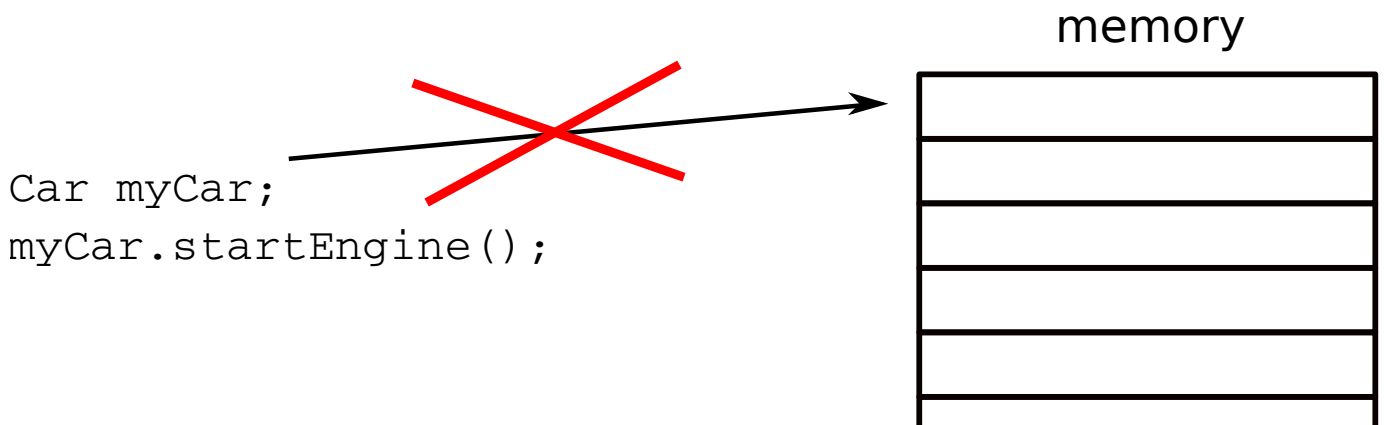
This will not compile: you'll get an error

`error: variable myCar might not have been initialized`

No memory has been allocated for it (using new); no object has been created.

If you do manage to fool the compiler into failing to notice that you're sending a message to an object that hasn't been initialised, then you'll get a runtime error: java.lang.NullPointerException

No reference, no memory allocated:



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;
```

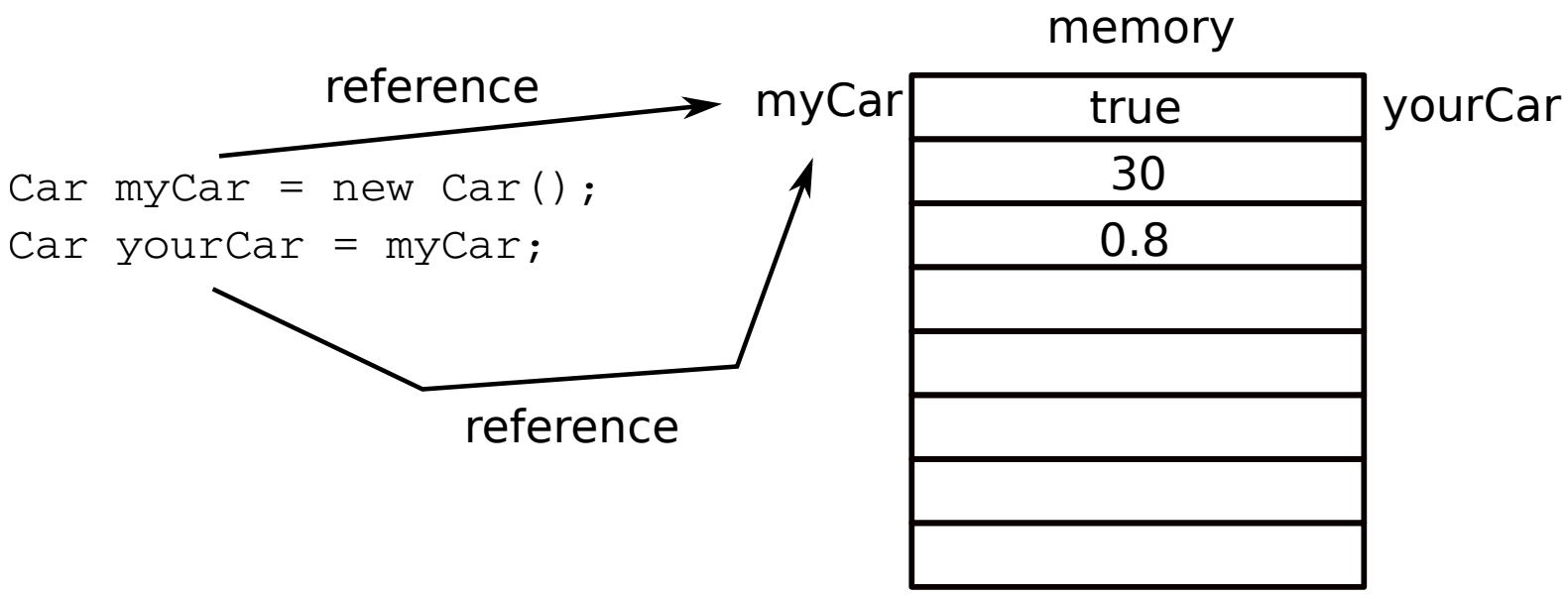
The null literal indicates an object reference pointing at nothing.

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!

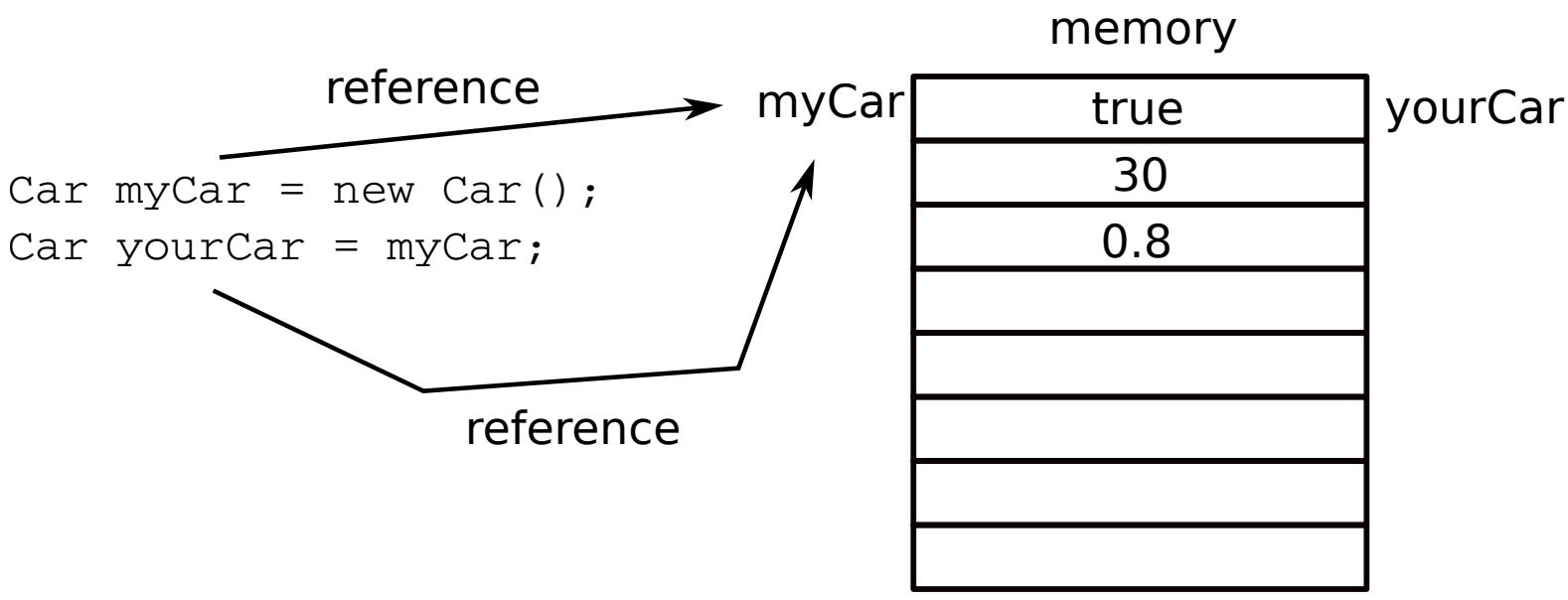
Class instances in memory

Copying an object instance:



Class instances in memory

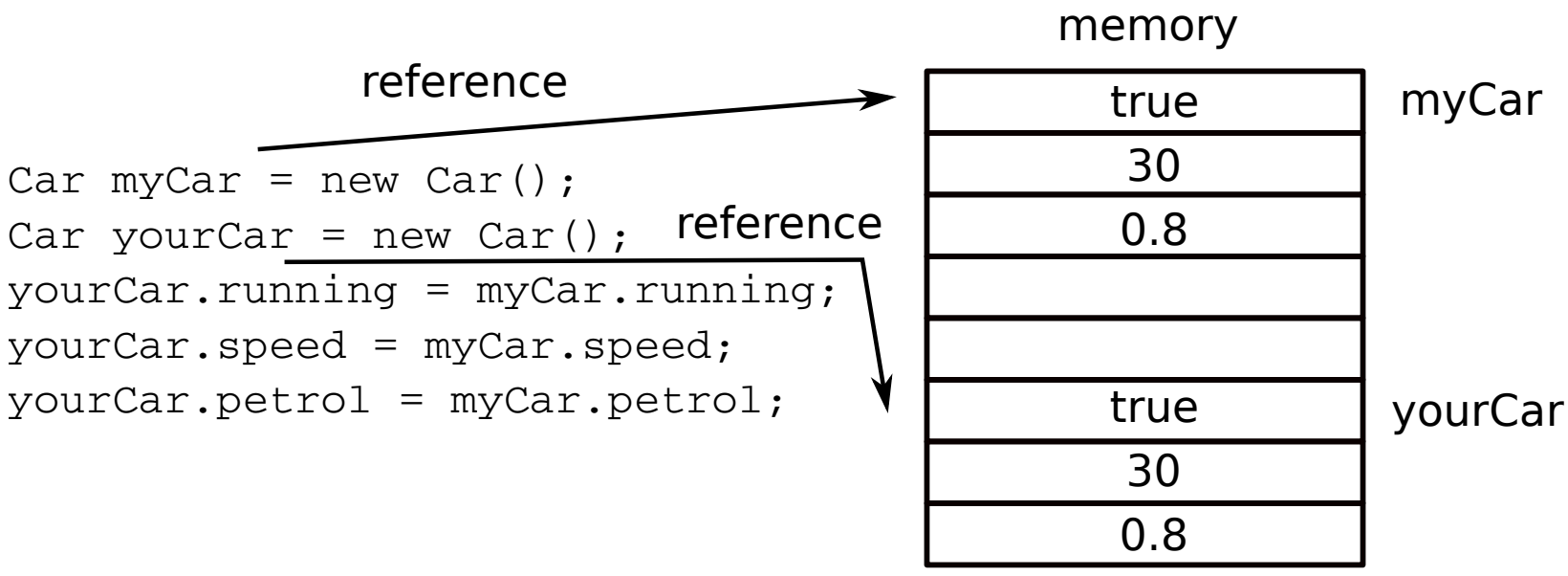
Copying an object instance:



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:

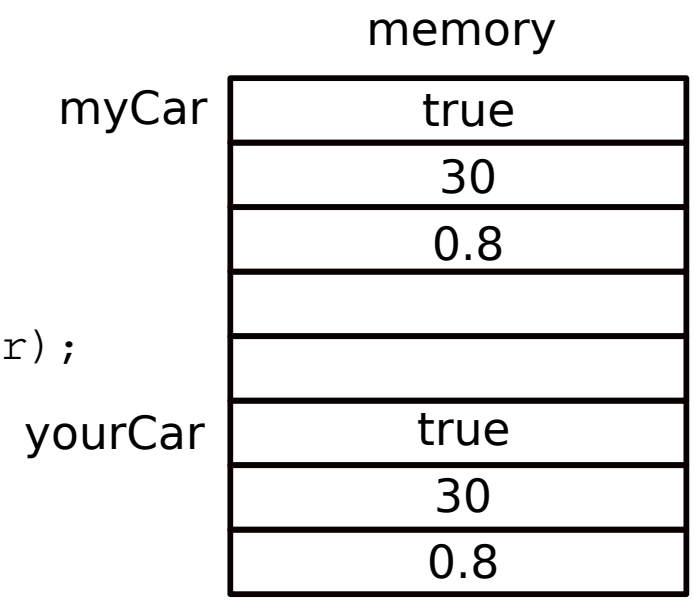


To copy an instance, a new one of the same type needs to be created and its entire state copied over.

Class instances in memory

Comparing class instances:

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

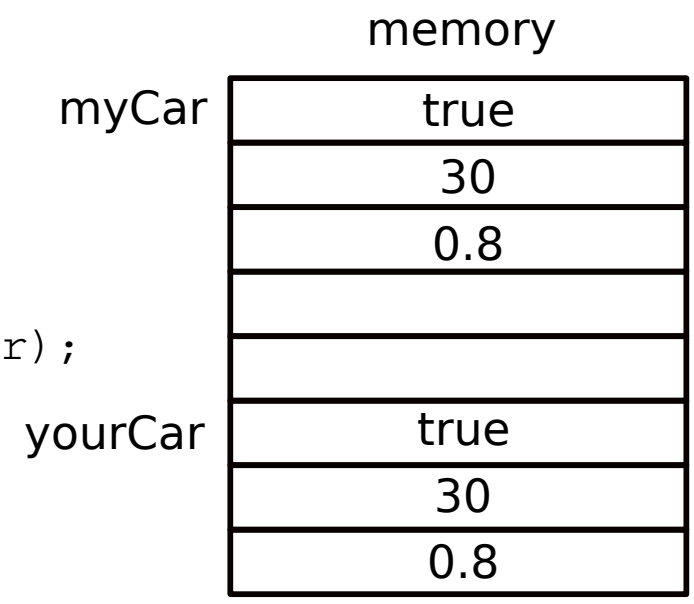


What does this print?

Class instances in memory

Comparing class instances:

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



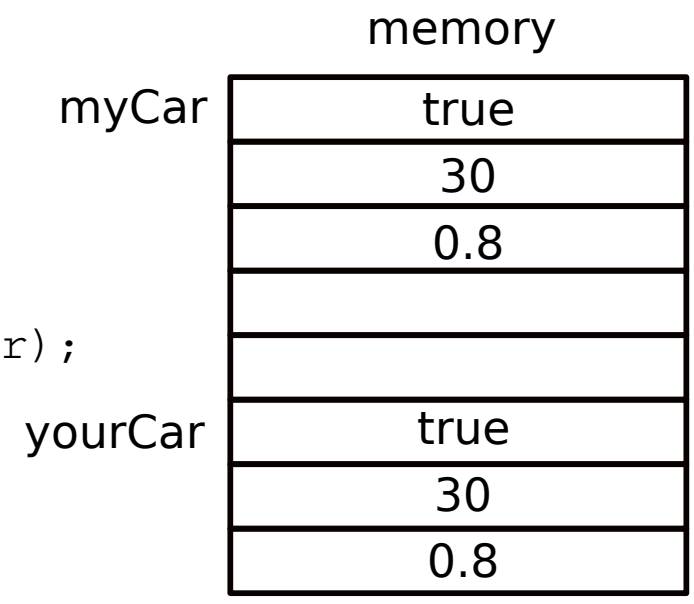
What does this print?

false

Class instances in memory

Comparing class instances:

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



What does this print?

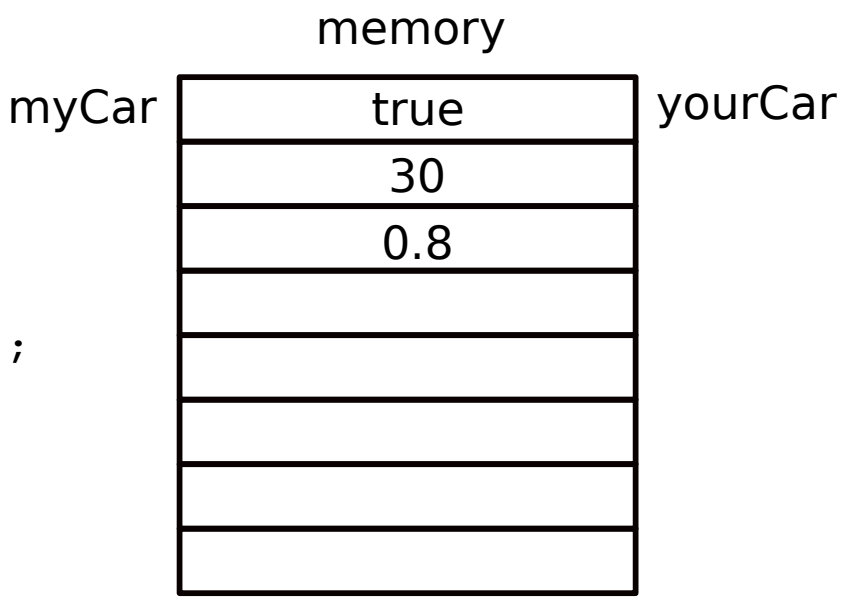
false

== 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);
```



What does this print?

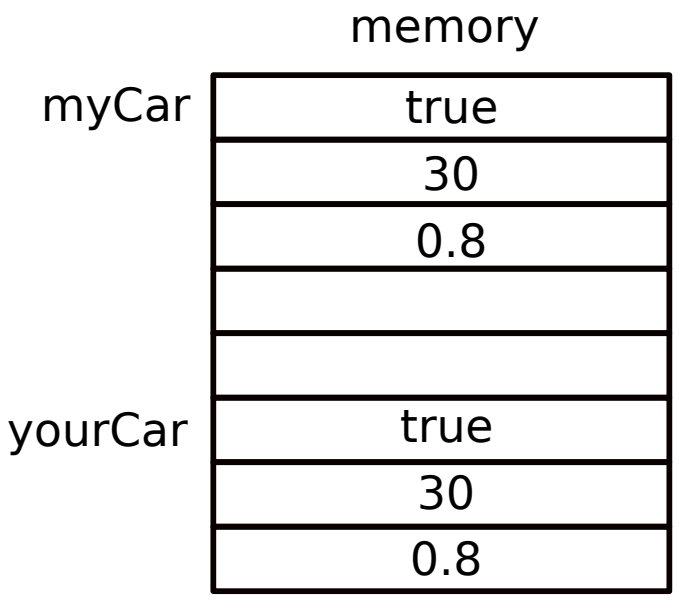
true

== 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);
```



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 Java library classes have sensible implementations of the comparison method **equals**.

```
String a = new String("hello ");
String b = new String("world");
String c = new String("hello world");

// prints true
System.out.println(c.equals(a+b));
```

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 practise that

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

What does it print?

```
public class ComparisonA {
  public static void main(String[] args) {
    int a = 5;
    int b = 5;
    System.out.println(a == b);
  }
}
```

What does it print?

```
public class ComparisonA {
  public static void main(String[] args) {
    int a = 5;
    int b = 5;
    System.out.println(a == b);
  }
}
```

Prints **true**. Values of primitive types are compared with ==.

What does it print?

```
public class ComparisonB {
  public static void main(String[] args) {
    String a = new String("hello␣");
    String b = new String("world");
    String c = new String("hello␣world");
    System.out.println(c == a+b);
  }
}
```

What does it print?

```
public class ComparisonB {
  public static void main(String[] args) {
    String a = new String("hello␣");
    String b = new String("world");
    String c = new String("hello␣world");
    System.out.println(c == a+b);
  }
}
```

Prints **false**. References of object instances are compared with ==.

What does it print?

```
public class ComparisonC {
  public static void main(String[] args) {
    String a = new String("hello␣");
    String b = new String("world");
    String c = new String("hello␣world");
    System.out.println(c.equals(a+b));
  }
}
```

What does it print?

```
public class ComparisonC {
  public static void main(String[] args) {
    String a = new String("hello␣");
    String b = new String("world");
    String c = new String("hello␣world");
    System.out.println(c.equals(a+b));
  }
}
```

Prints **true**. States of object instances are compared with `equals`.

Wrapper classes for primitive types

Primitive types were originally included in Java essentially for efficiency. However, some things can only be done with objects, not with instances of primitive types.
E.g. Java provides many kinds of collections and only objects can be placed in collections.
Therefore, for each primitive type there is a wrapper class. This lets you create an object which simply wraps up a primitive type element.
In early versions of Java we used to write things like

```
Integer i = new Integer(7); // NOW DEPRECATED!
```

Autoboxing and Unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.

```
Integer num = 5;
```

If the conversion goes the other way, this is called **unboxing**.

```
Integer num = myObject.methodReturningInteger();
int sum = 10 + num;
```

These days you will seldom have to think about these conversions – they will mostly just happen.

What does it print?

```
public class ComparisonD {
  public static void main(String[] args) {
    Integer a = 5;
    Integer b = 5;
    System.out.println(a == b);
  }
}
```

What does it print?

```
public class ComparisonD {
  public static void main(String[] args) {
    Integer a = 5;
    Integer b = 5;
    System.out.println(a == b);
  }
}
```

Prints **true**. Even though object references are compared, true is printed because the literal 5 is cached by the compiler and the same object is used under the hood.

This caching process of certain literal values is called **Interning**.

What does it print?

```
public class ComparisonE {
  public static void main(String[] args) {
    Integer a = 200;
    Integer b = 200;
    System.out.println(a == b);
  }
}
```

What does it print?

```
public class ComparisonE {
  public static void main(String[] args) {
    Integer a = 200;
    Integer b = 200;
    System.out.println(a == b);
  }
}
```

Prints **false**. Integer literals are only cached from -128 until 127 (1 byte).

What does it print?

```
public class ComparisonF {
  public static void main(String[] args) {
    String a = "this␣is␣a␣test";
    String b = "this␣is␣a␣test";
    System.out.println(a == b);
  }
}
```

What does it print?

```
public class ComparisonF {
  public static void main(String[] args) {
    String a = "this␣is␣a␣test";
    String b = "this␣is␣a␣test";
    System.out.println(a == b);
  }
}
```

Prints **true**. String literals are also interned.

What does it print?

```
public class ComparisonG {
  public static void main(String[] args) {
    String a = new String("this␣is␣a␣test");
    String b = new String("this␣is␣a␣test");
    System.out.println(a == b);
  }
}
```

What does it print?

```
public class ComparisonG {
  public static void main(String[] args) {
    String a = new String("this␣is␣a␣test");
    String b = new String("this␣is␣a␣test");
    System.out.println(a == b);
  }
}
```

Prints **false**. If you explicitly use a constructor, two different object instances are created.

How to compare things in Java

For Primitives use ==

For Objects

- use == if you want to know whether two references refer to the very same object
- but usually, use `equals`.

By the magic of inheritance, which we'll come, to, every object in Java understands `equals`.

The writer of the class may have implemented the `equals` method so that it compares the states of two objects of that class in a way that makes sense for that class.

If not, there is a default implementation which is just ==.

== implies `equals` but not vice versa

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 that refers to 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 that refers to 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 class method requires the class name: `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 – especially useful for constants.

Global Constants

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

```
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 over the world. Why?

Summary: Why use object orientation?

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

Summary: Why use object orientation?

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

- 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

Java Tutorial

does things in a rather different order from us. You could read to the end of Chapter 4, but you will meet things we have not covered yet.

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.