

Object Interaction

Buying a car

Consider a car as a **whole** object and think about properties such as:

- colour
- fuel efficiency
- capacity
- ...

Building a car

Consider the **parts** and how they interact:

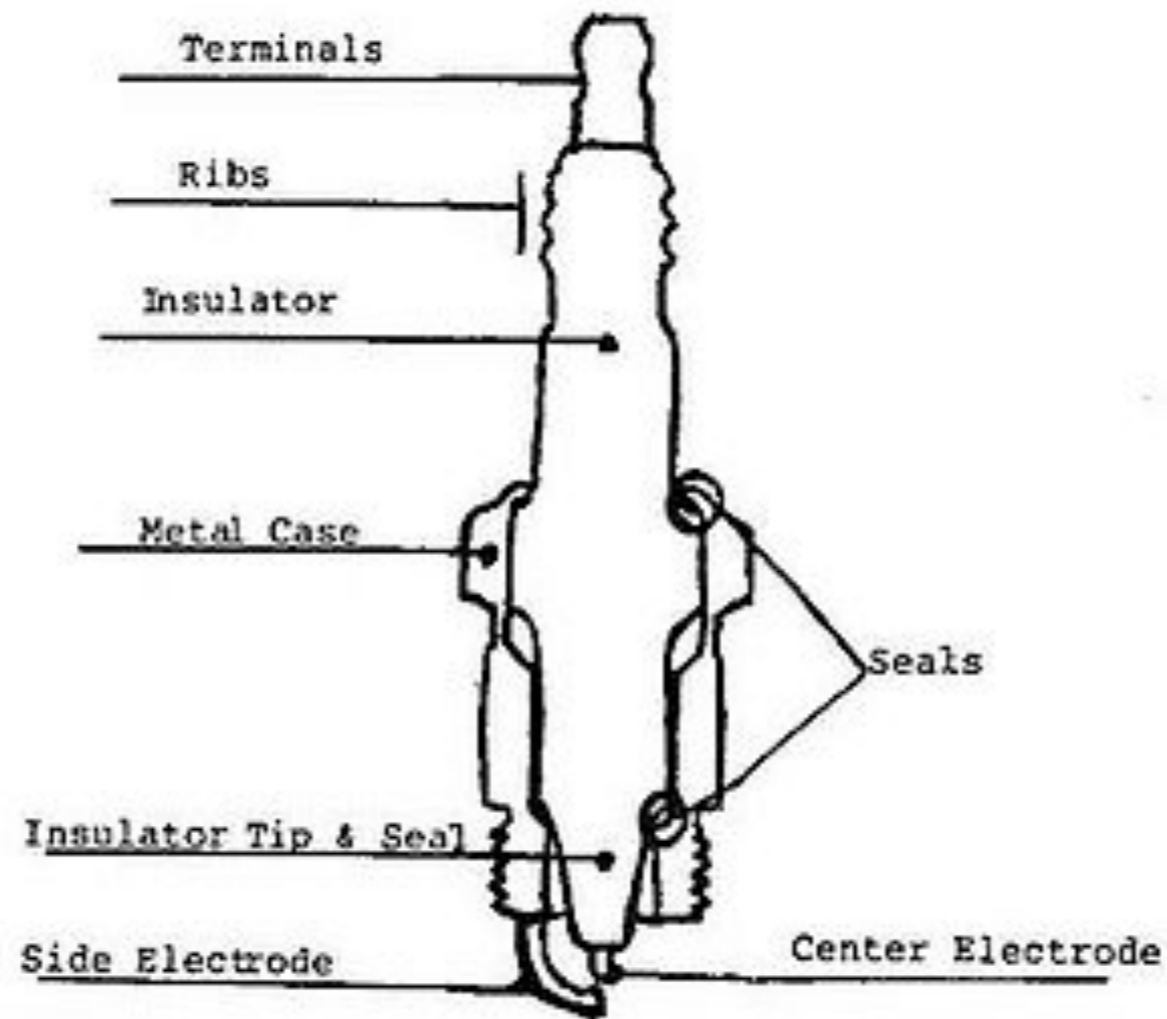
- tyres
- engine
- chassis
- gear-box

Building an engine

If we were to build our own engine:

- manifold
- distributor
- radiator
- ...

Building a spark-plug



Abstraction

To implement a large complex we break it down into parts (and subparts).

Each part has:

- a **public interface** that describes how it can be **used**
- a **private implementation** that describes how it **works**

Abstraction

“Chunking” a problem like this is called **abstraction**.

It makes problem-solving easier by allowing us to work at the appropriate **level of detail** for the particular job at hand.

Abstraction

To implement abstraction in our programs we create multiple classes which represent different parts of the problem at different levels of detail.

The more abstract classes are then implemented using the more detailed classes.

Primitive types

At the bottom of the abstraction hierarchy are the **primitive types** like **int** and **boolean**.

These types are the simplest units of data represented in Java.

String is not a primitive type. It is actually a class implemented in terms of **int**, but those details are hidden.

Classes are types

Classes are data-types and can be used in the same way as primitive types:

- to define variables (fields, parameters)
- as method return types

Classes as fields

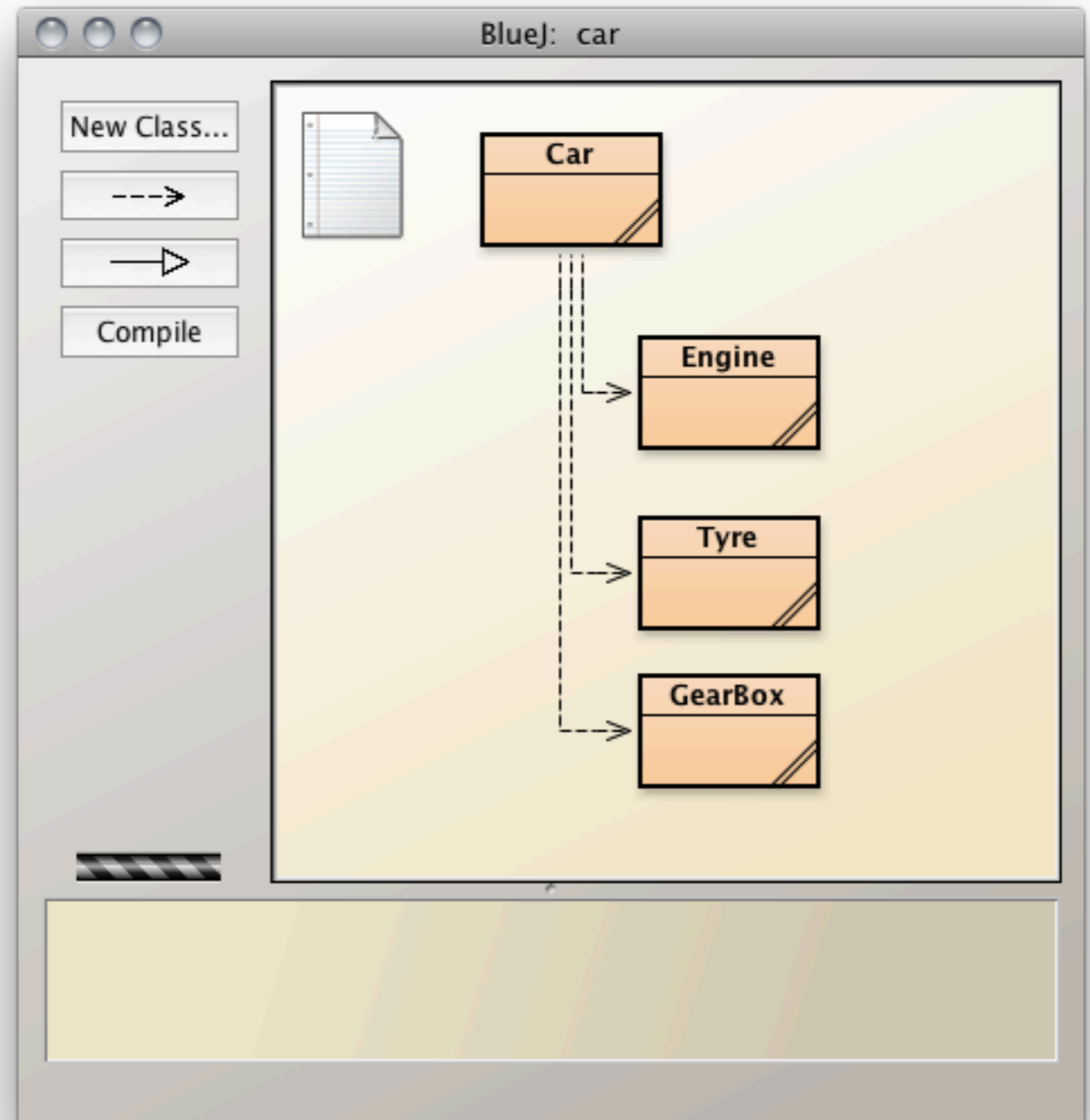
The syntax for defining a field which is a class is the same as for a primitive type:

```
public class Car
{
    private Engine myEngine;
    private GearBox myGearBox;
}
```

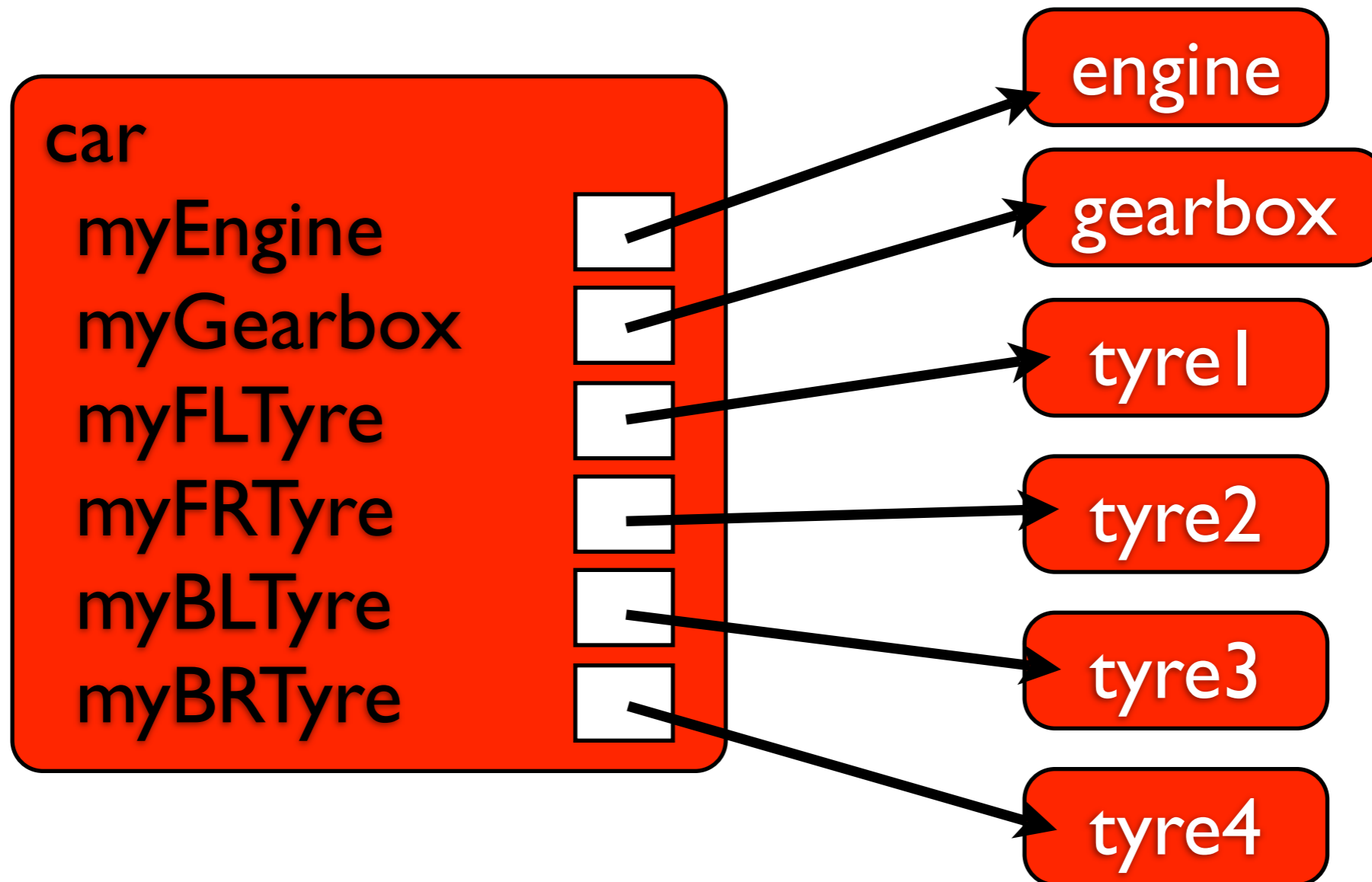
Class diagrams

The Car class uses:

- Engine class
- Gearbox class
- Tyre class



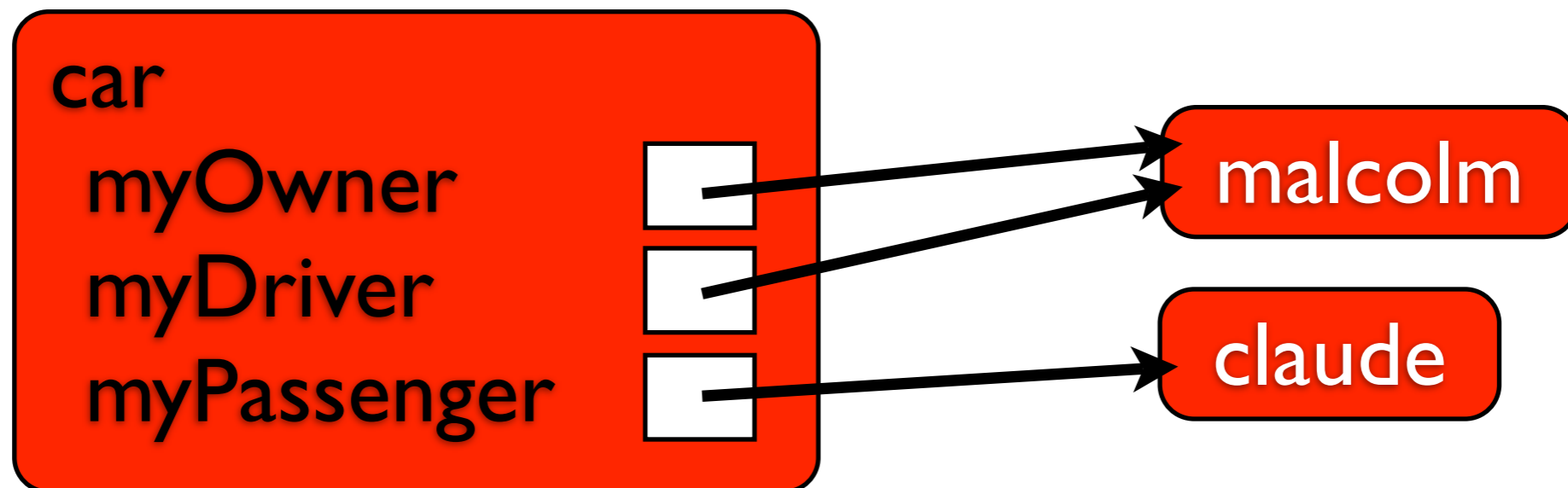
Object diagrams



Object references

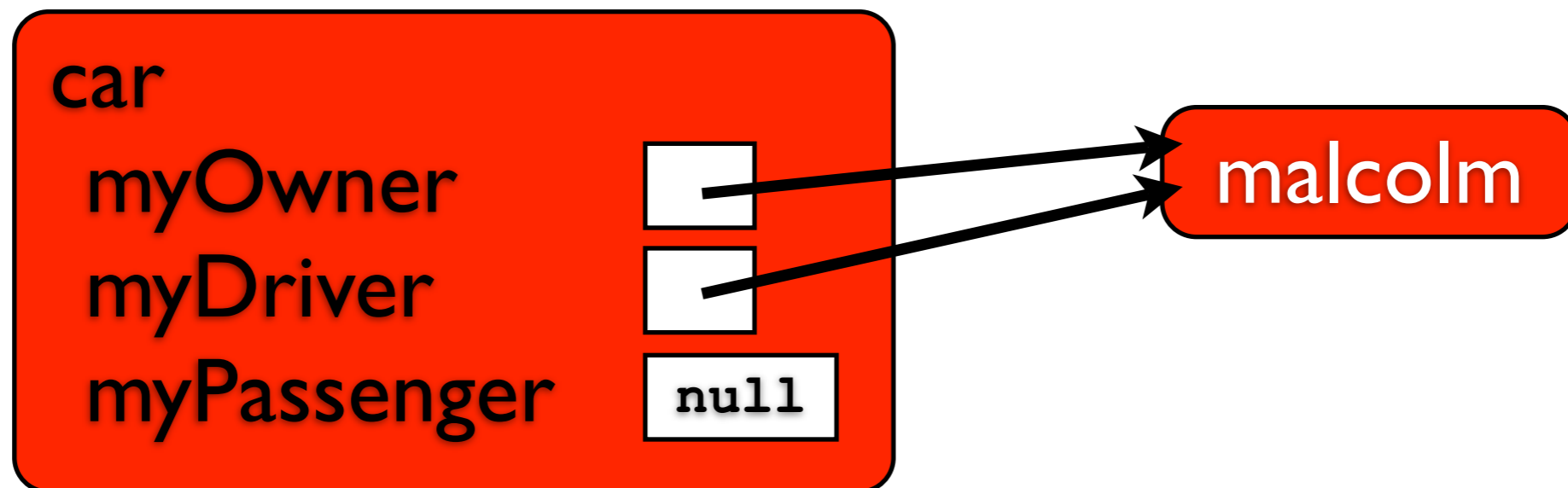
Variables contain **references** to objects (arrows) not the objects themselves.

Two variables can both refer to the same object.



Null references

A variable can also contain **no object**. In this case we use the special **null** value to represent a **null reference**.



Constructing objects

We can construct objects in code using the **new** statement:

```
public Car()  
{  
    myEngine = new Engine();  
}
```

This calls the **Engine()** constructor on the **Engine** class to initialise the new object.

Multiple Constructors

Classes can have several constructors with different parameters:

```
public Engine()  
{  
    // default capacity  
    myCapacity = 1600;  
}  
  
public Engine(int capacity)  
{  
    // user specified  
    myCapacity = capacity;  
}
```

Multiple Constructors

Which constructor is used depends on the parameters given:

```
// use Engine() constructor  
myEngine = new Engine();
```

```
// use Engine(int) constructor  
myEngine = new Engine(1200);
```

Method calls

A **method call** is another kind of **statement**.

It invokes another method.

When the invoked method **finishes**, execution continues at the **next statement**.

Method calls

Method calls can be:

- **internal** - calling methods on the **same** object
- **external** - calling methods on a **different** object

Internal methods

If we use the sequence of instructions **repeatedly** in a class, it is a good idea to separate it into a **private method**.

This has two advantages:

- **abstraction** - the code is easier to read
- **centrality** - changes to the code are made in just one spot.

Internal methods

```
private void updatedisplay()  
{  
    // fields have changed  
    // update the display  
}  
  
public void setHours(int hours)  
{  
    myHours = hours;  
    updatedisplay();  
}
```

Method call syntax

The syntax for an **internal** method call:

```
// invoke the method
```

```
methodName (param1 , param2) ;
```

```
// execution continues here
```

Method parameters

When we call a method, we must provide **parameter values** with **types** that match the method parameters.

Method parameters

```
private void say(String message,  
                 int volume)  
{  
    // say something  
}
```

Correct

```
say("Hello", 1); // values
```

```
say("Hi", 3 * 5); // expressions
```

```
String name = "Malcolm";
```

```
int volume = 4;
```

```
say(name, volume); // vars
```

Incorrect

```
say("Hello"); // missing int
```

```
say(2, "Hi"); // wrong order
```

```
say(true, 4); // wrong type
```

Private methods

We use the `private` access modifier to indicate that the method is for `internal use only`.

External methods

We use external methods to perform actions on other objects.

Only **public** methods can be accessed externally.

Method call syntax

The syntax for an **external** method call:

```
SomeObject obj =  
    new SomeObject();  
  
// invoke the method  
obj.methodName(  
    param1, param2);  
  
// execution continues here
```

Return values

If a method has a **return value** we can use it as an **expression**.

```
TicketMachine machine  
    = new TicketMachine(100);  
  
int price = machine.getPrice();  
  
// price is now 100
```