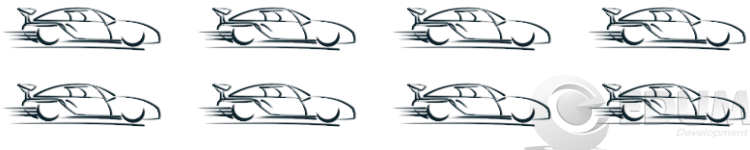
# Generic types and collections

## Generic lists

A car dealership wishes to use the **Car** class to keep track of sales records.

They need to store data about the cars they have **for sale**.

There's no **upper limit** on the amount of cars they might have for sale.



## Generic lists

We could use an array:

```
Car[] carsForSale = new Car[???];
```

When initializing the array, we have to provide a size

We could solve this in a number of ways, such as limiting the number of cars we can have.

```
Car[] carsForSale = new Car[100];
```

However, there's a much simpler solution.



## Generic lists

We have access to something called a **List**.

In order to use it, we must have to import the following packages:

```
import java.util.List;
import java.util.ArrayList;
```

## Generic lists

Then, we can create a **ArrayList** in the following way:

```
List<Car> carsForSale = new ArrayList<Car>();
```

Generic type argument.

From Java 7 they added so that we don't have to repeat the type on the right hand side.

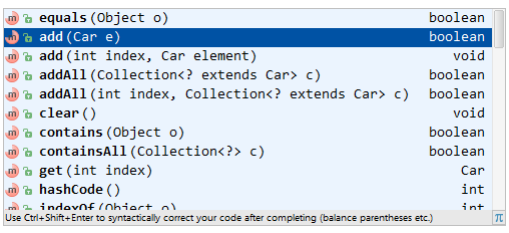
```
List<Car> carsForSale = new ArrayList<>();
```

This is called the diamond operator


## Creating the List object

An actual List class containing **Car** objects is created, and we get **IntelliSense** as well:

```
carsForSale.
```




# Modifying the List (1)

In order to add a Car to the list, we use the call **add** method:

```
public static void main(String[] args) {
    Car oldRustyFord = new Car(4,4,"Ford Sierra", "Blue");
    Car disgracefulVolvo = new Car(4,4,"Volvo 240", "Unknown");
    Car oldKoreanCar = new Car(4,4,"Kia", "Green");
    Car smallNissan = new Car(4,4,"Nissan Almera", "Red");

    List<Car> carsForSale = new ArrayList<>();

    carsForSale.add(oldRustyFord);
    carsForSale.add(disgracefulVolvo);
    carsForSale.add(smallNissan);
    carsForSale.add(oldKoreanCar);
}
```


# Modifying the List (2)

In order to remove a Car from the list, call **remove**.

```
public static void main(String[] args) {
    Car oldRustyFord = new Car(4,4,"Ford Sierra", "Blue");
    Car disgracefulVolvo = new Car(4,4,"Volvo 240", "Unknown");
    Car oldKoreanCar = new Car(4,4,"Kia", "Green");
    Car smallNissan = new Car(4,4,"Nissan Almera", "Red");

    List<Car> carsForSale = new ArrayList<>();
    carsForSale.add(oldRustyFord);
    carsForSale.add(disgracefulVolvo);
    carsForSale.add(smallNissan);
    carsForSale.add(oldKoreanCar);

    // Removing
    carsForSale.remove(oldRustyFord);
}
```

# Encapsulating a list


## Creating a **CarStore** class (1)

Keeping everything in the main method is seldom sensible.

We might want to **encapsulate** this in a **class** that we can reuse.

We can use any type as a **field**, including **Lists**.


# Creating a **CarStore** class (2)

Encapsulate everything in a class called **CarStore**:

A list as a private field, inaccessible from outside this class

Constructor, initializing the list

Methods delegating calls to the carsInStore field

A get-only method, delegating to the carsInStore field

```
public class CarStore {
    private List<Car> carsInStore;

    public CarStore() {
        carsInStore = new ArrayList<>();
    }

    public void addCarToStore(Car car) {
        carsInStore.add(car);
    }

    public void removeCarFromStore(Car car) {
        carsInStore.remove(car);
    }

    public int numberOfCarsInStore() {
        return carsInStore.size();
    }
}
```


# Using the **CarStore** class

This class can now be used throughout the application:

```
CarStore store = new CarStore();
Car oldRustyFord = new Car(4, 4, "Ford Sierra", "Blue");
Car disgracefulVolvo = new Car(4, 4, "Volvo 440", "White");

// Adding
store.addCarToStore(oldRustyFord);
store.addCarToStore(disgracefulVolvo);

// Removing
store.removeCarFromStore(oldRustyFord);
```

Creating reusable objects is a key property of good object oriented design.

## Reviewing the **CarStore** class (1)

The **CarStore** class doesn't really contain any logic of its own.

It merely delegates to the **List**, so why not just pass the **List** around?



## Reviewing the **CarStore** class (2)

First of all, the **CarStore** class acts as an abstraction layer between the **List** and any class using it.

If we want to add a certain **behaviour** to our store, this class works as an **extension point**.

Secondly, we gain a **semantic** feature: our method names are described in more detail to suit our class (**addCarToStore** instead of **add**).

# More List features

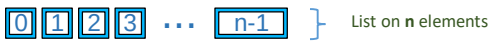

## More **List** features (1)

The elements of an **ArrayList** can be accessed by an index:

```
List<Car> carsInStore = new ArrayList<>();

// Add some cars
Car oldRustyFord = new Car(4, 4, "Ford Sierra", "Blue");
Car disgracefulVolvo = new Car(4, 4, "Volvo 440", "White");
store.addCarToStore(oldRustyFord);
store.addCarToStore(disgracefulVolvo);

// Get the first car
Car myCar = carsInStore.get(0);
System.out.println(myCar.getCarMake());
```




The index always starts at 0!

# More **List** features (2)

A **List** can also be iterated using a **for-each** loop (as well as an ordinary **for** loop using the **get** method)

```
List<Car> carsInStore = new ArrayList<>();

// Add some cars
Car oldRustyFord = new Car(4, 4, "Ford Sierra", "Blue");
Car disgracefulVolvo = new Car(4, 4, "Volvo 440", "White");

for (Car car : carsInStore) {
    System.out.println(car.getCarMake());
}
```

Being able to use for-each syntax is due to something called iterators.


# Exercise 16

Let's do exercise 16