Having seen a skeleton of an inheritance model, we can look at the full code for a trivial example of object modelling using composition. Look out for the keyword `new` which indicates that we are creating an instance of the class `Address`.

This is a working example, albeit a trivial one. The files can be compiled separately or together, but keep them in the same directory:

```java
public class Address {
    private String street;
    private String city;
    private String county;
    private String country;

    // established data storage, now construct class
    // passing it some values as parameters
    public Address(String s, String c, String co, String cou) {
        street = s;
        city = c;
        county = co;
        country = cou;
    }

    // provide an accessor method to pass data with
    public String retrieve() {
        return street + "," + city + "," + county + "," + country;
    }

} // end class definition
// This is the main class, which declares an instance of address:

import java.io.*;
public class Contact {
    public static void main(String[] args) {
        String name = "Joe Soap";

        // create new object
        Address a = new Address("10 Downing Street",
            "Westminster", "London", "England");
        System.out.println("Name=" + name + "Address=" +
            a.retrieve());
    }
}
```

It should be possible to see, from this trivial example, that a bigger application might be a contact book, which could include classes for name, address and telephone numbers. The general rule when designing classes is that if you have a large number of variables to deal with and some of them are of the same category, such as the lines of an address, then you have a candidate for a further class. The advantages of this approach, which may not be immediately apparent in such a trivial example, are that the details of the address class are hidden from the class that uses it. This means that, if we decide to alter the way in which the display is put onto the screen, we know that all of the code that we need to alter will be found in one place — the `retrieve()` method of the `Address` class. The internal details of the `retrieve()` method matter not a jot to the `Contact` class, so we need alter nothing else in the whole program. If for a moment we revert to our previous example of aeroplanes and engines, the benefit should be obvious: we have a single implementation of an engine which might be used by many different aeroplanes. By upgrading that one class, we immediately upgrade any object that is instantiated using that engine — i.e. the whole airforce!

The relationships between classes we have looked at so far include composition and inheritance. There are also circumstances where we would like different classes to behave in partially similar ways. In Java this is dealt with by the use of `interfaces`.

### 3.2.1 Class relationships

- **Use** — when a class uses the public methods of another class.
- **Inheritance** — when a class is derived from another class.
- **Composition** — when a class is composed of other classes.