Exercise 11: Abstract base classes

In the last exercise, we worked out the difference between base classes (`extends`) and interfaces (`implements`). Since the different brakes (disc, drum and cantilever brakes) are mechanically very different but nevertheless can all basically brake, we chose an interface here. This had the disadvantage that the individual brake variants implement the Brake interface, but do not extend it. However, if we now want to specify attributes such as `manufacturer` for all brakes, then there are two options:

a) Conversion of the interface into a class with the corresponding attributes

```
public class Brake {
    private String manufacturer;
    public Brake(String m) {
        this.manufacturer = m;
 }
    public void brake() {
 }
 }

public class DiscBrake extends Brake{
    public DiscBrake(String m) {
        super(m);
 }
 }
```

b) Extension of the interface with appropriate getter/setter methods

```
public interface Brake {
    void brake();
 string getManufacturer();
    void setManufacturer(String m);
 }

public class DiscBrake implements Brake{
    @Override
    public void brake() {}

    @Override
    public string getManufacturer() {
        return zero;
 }

    @Override
    public void setManufacturer(String m) {}
 }
```

The first version (a) has the disadvantage that we now have to implement the brake method in `Brake` without knowing which type of brake it is -- if that makes semantic sense at all! The second version (b) has the disadvantage that in every realisation of Brake, a separate variable must be created for the manufacturer.

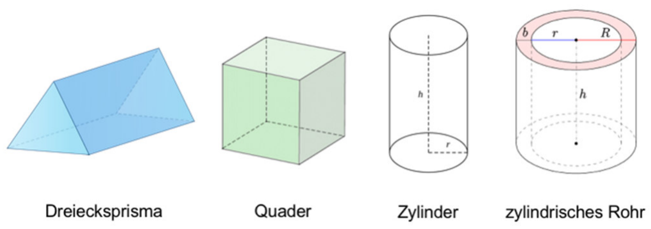
The solution is an abstract base class (abstract class), in which the `brake` method is marked as abstract and is not implemented (not `{ ... }`, but `;`)

Task 1: The abstract base class Brake

We have part of the sample solution of the last task in which `Brake` is an interface.

- Draw a UML diagram. To do so, model Brake as an abstract base class, which models `Manufacturer` (`String`) and `SerialNumber` (`String`) as private attributes and prescribes the methods `void brake()` and `void needsService()`.
- Adjust the given implementation to match the UML diagram. Use fictitious values for manufacturer and serial number.
- Make sure that all tests in `BrakeTests` run successfully.

Task 2: Three-dimensional shapes



The triangular prism, cuboid, cylinder and cylindrical tube shapes shown above are simple volumes which are calculated from the base area times the height. Model the four shapes using a common base class.

The class files Volumes.java, TriangularPrism.java, Cuboid.java, Cylinder.java and CylindricalTube.java are already given.

- Draw a UML diagram that places the classes in appropriate inheritance relationships.
- A volume should use the double volumes() method to calculate the actual volume (i.e. base area times height), but code duplication should be avoided.
- Make clear what values are needed to calculate the volume, and in which class these can be specified.
- Complete the given class bodies according to your diagram and implement the methods.
- Make sure that the tests in `VolumesTests` are successful.

Information:

- Use floating-point numbers for all calculations.
- The triangular surface can be calculated from the three side lengths using Heron’s formula.
- The cross-section of a tube can be calculated with the inner and outer radius.
- An abstract base class is required which has both specific and abstract methods.
- With enough preliminary consideration we can work on a top-down basis, but bottom-up may be easier. To do this, first implement the logic for two classes, in order to then isolate similarities.