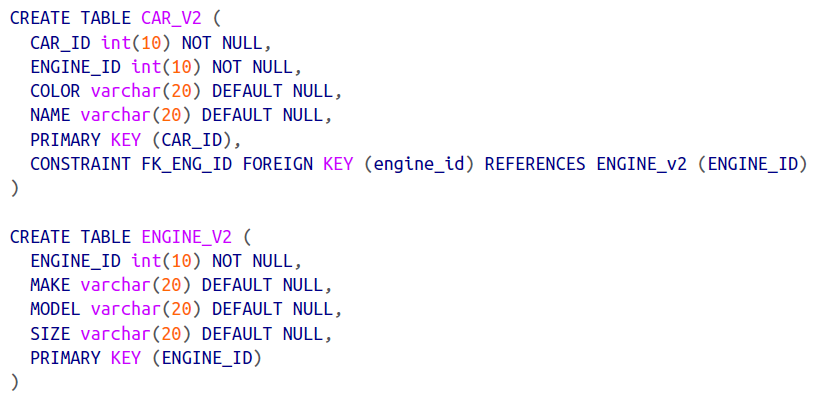
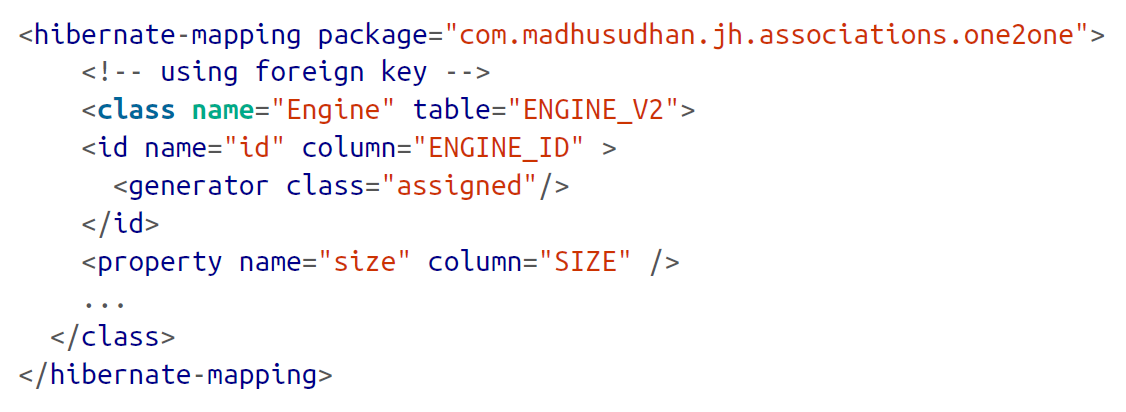
**Using a Foreign Key**

To use the foreign key strategy, we have to alter the table and mapping definitions (the POJOs remain the same). Modified table definitions are given here (note that the table names have been modified with suffix V2 so they won’t clash with our earlier incarnations):

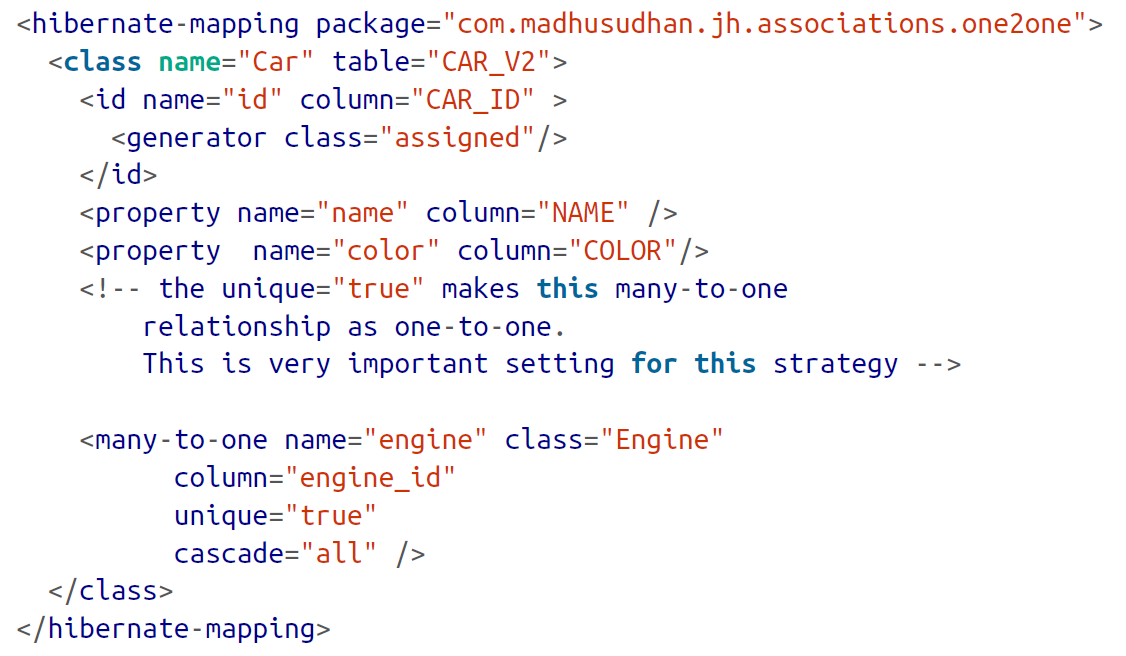


The *Engine\_V2* definition is simple and sweet; it’s a normal table with a primary key *ENGINE\_ID.* The notable change is in the CAR\_V2 table: in addition to having its own primary key (CAR\_ID), it also has a foreign key (ENGINE\_ID) that points to the ENGINE\_V2 table.

The next step is to define the mappings. The *Engine* mapping shouldn’t be any surprise to you:



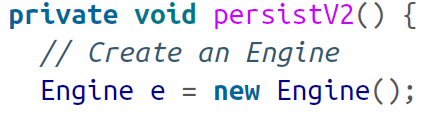
We define the application strategy for setting the id of the object and map the properties of the class to the relevant table columns. The next snippet is the mapping for the Car object:

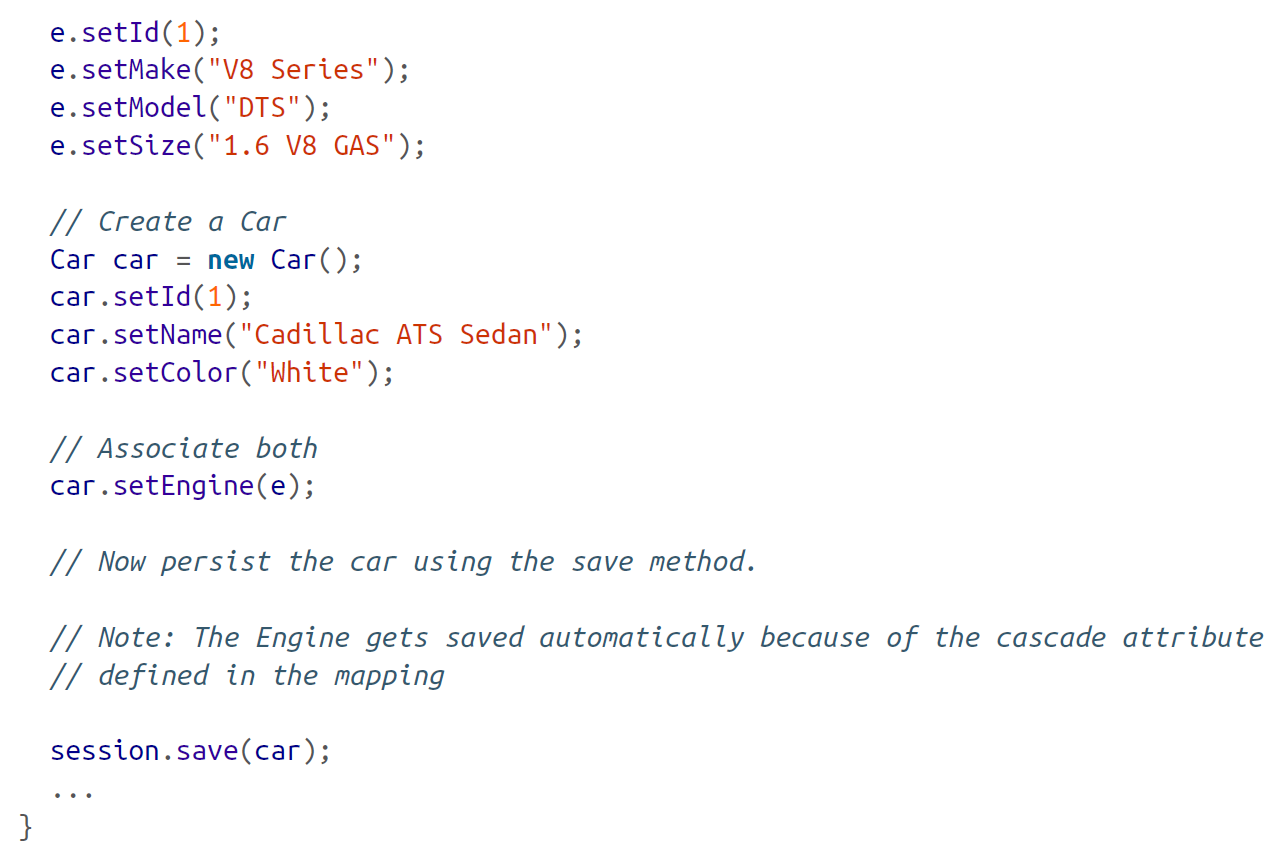


Note the use of the many-to-one element (instead of the one-to-one element we’ve used in the earlier case) with the additional attribute *unique=”true”.* By setting this attribute, we are essentially converting a many-to-one relationship to a one-to-one association.

This attribute will ensure that one Car is associated to one *Engine.*

Let’s modify the test client to reflect the changes:





From the preceding code, you can see that we are creating both entities, *Engine* and *Car,* respectively. We then add an engine to the car and save it to the database. Notice that we are not saving the engine explicitly. That’s because, when the car is persisted, its associated objects (engine, in this case) are saved too because of the *cascade=”all”* attribute defined in the mapping file.

Before we wrap up the discussion on one-to-one association, you may be wondering what’s the preferred approach? I personally prefer the foreign key relationship by using the many-to-one element with the unique attribute set to true. In fact, Hibernate recommends this approach too.